XML / WEB SERVICES

&

RESTful Services

(XML, DTD, XSD, JAX-P, JAX-B, SOAP, WSDL, JAX-WS, SOAP UI, JAX-RS & POSTMAN)

MR. Ashok

Facebook Group: Ashok IT School

Email: ashok.javatraining@gmail.com

INDEX

Webservices Introduction

Introduction ----------1

Distributed Programming----------2-3

CORBA----------4

RMI----------5-6

EJB----------7

Common challenges in Distributed Programming----------8

Webservices specifications----------9

Webservices Architecture----------10-11

Motivation and Characteristics----------12-13

Core Webservices Standards----------14-16

XML ----------18-22

DTD

Introduction----------24

Simple Elements----------25

Compound Elements----------26

Attributes----------27

DTD Types----------28-29

DTD Drawbacks----------30

XSD

Introduction----------32

Simple Elements Declaration----------33

Compound Elements Declaration----------34-35

Linking XSD to XML----------36

Element Cardinality----------37

Compositors (sequence, all and choice) ---------- 38-39

Global Elements----------40

Group Elements----------41-42

Inheritance in XSD----------43

Attributes in XSD----------44-46

Restriction on Simple Types----------47-51

XSD Namespaces----------52-53

xs: include----------54

xs: import----------55-56

JAX-P

Introduction---------- 58

SAX Parser---------- 59-64

DOM Parser---------- 65-69

STAX Parser----------70-71

XML validation against XSD----------72-73

JAX-B

Introduction----------74

JAX-B Architecture----------75

Binding----------76-77

One-Time Operations----------78

Using XJC----------79-80

Runtime Operations----------81

Un-Marshalling----------81-83

Marshalling----------84

In-Memory Validation----------85

Getting Started with Webservices

Introduction----------86

Webservices pre-requisites----------87-88

JAX-WS

Introduction----------90-94

JAX-WS Features----------95

JAX-WS Implementations----------96

Provider Development (Contract Last Approach) ----------97-100

Consumer Development----------101 - 102

Provider (Contract First Approach) ----------103-107

SOAP Handlers Introduction----------108

Provider with SOAP Handler----------108-117

Consumer with SOAP Handler----------118-121

APACHE AXIS2

Introduction ----------122

Environment Setup----------123-125

webservices development using AXIS2----------126-131

Spring + JAX-WS Integration----------132-138

Restful services

Introduction----------140-142

Http Methods----------143-149

JAX-RS API Introduction----------150

JSON----------151-155

GSON API----------156-157

Jersey----------158-159

JAX-RS Annotations overview----------160

Restful service first application----------162-164

JAX-RS Injection----------165-167

Rest Resource with JSON data----------168-170

@Consumes----------171-173

Jersey 2.x application----------174-179

Jersey Client----------180-183

Rest App without XML----------185-186

Rest Easy Introduction----------187

Restful Service with RestEasy----------188-189

RestEasy Client----------190

Bootstrapping in RestEasy----------191-192

File Uploading----------193-194

Spring + Restful Service Integration----------195-211

Response ----------212

Http Headers ----------213--214

Extracting Request Parameters----------215-218

Exception Handling ----------219-220

Caching ----------221-224

Security ----------224-230

Conclusion----------231-233

POSTMAN----------234-238

Spring Integration with REST----------239-247

Web Services are the mechanism or the medium of communication through which two applications / machines will exchange the data/business services irrespective of their underline architecture and the technology.

In general, software applications are developed to be consumed by the human beings, where a person sends a request to a software service which in-turn returns a response in human readable format.

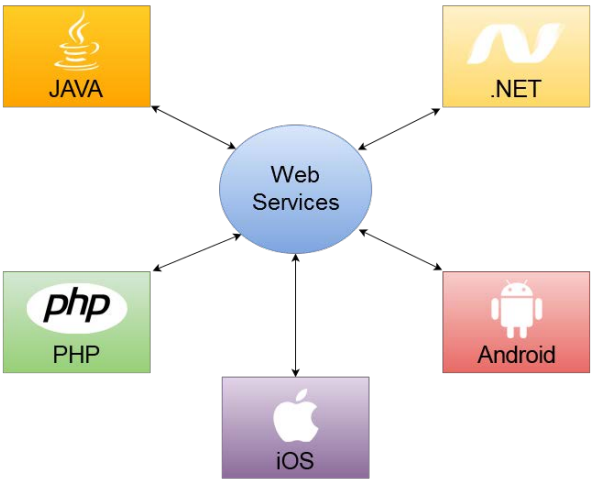
In the modern era of technology if we want to build a software application we don't need to build each and everything from scratch. There are lots of readymade services available which we can plug into our application and we can start providing those services in our application.

For example, we want to display weather forecast information in our application but weather forecast information will not be available in our application database. So, our application should talk to Satellites to get weather forecast information. So, do we really need to write the complex logic to talk with satellite to get the details? - No, we don't need to collect, process and render the data in your application from satellite. Because some applications are already developed by people to get the weather forecast details from satellites so we can buy the services from those people who already well-established in processing and publishing such kind of data.

Now the question is how our application can talk to weather forecaster application? That's where Web services comes into picture. Webservices allow us establish communication between two applications irrespective of the language and irrespective of the platform. If two applications which are developed in two different languages able to communicate with each other, then those applications are called as **Interoperable applications**. Webservices are used develop interoperable applications.

A Java application can connect with .Net application using Webservices and vice versa

A PHP application can connect with java application and vice versa



Before jumping into Webservices development first let's see some background related to Distributed Computing.

The Internet has revolutionized our business by providing an information highway, which acts as a new form of communication backbone. This new information medium has shifted business from the traditional brick-and-mortar infrastructures to a virtual world where they can serve customers not just the regular eight hours, but round-the-clock and around the world. Additionally, it enhances our organizations with significant benefits in terms of business productivity, cost savings, and customer satisfaction. As a result, modern organizations are compelled to re-evaluate their business models and plan on a business vision to interact with their customers, suppliers, resellers, and partners using an Internet-based technology space. To achieve this goal of obtaining an Internet business presence, organizations are exposing and distributing their business applications over the Internet by going through a series of technological innovations. The key phenomenon of enabling business applications over the Internet is based on a fundamental technology called distributed computing.

Distributed computing has been popular within local area networks for many years, and it took a major step forward by adopting the Internet as its base platform and by supporting its open standard-based technologies.

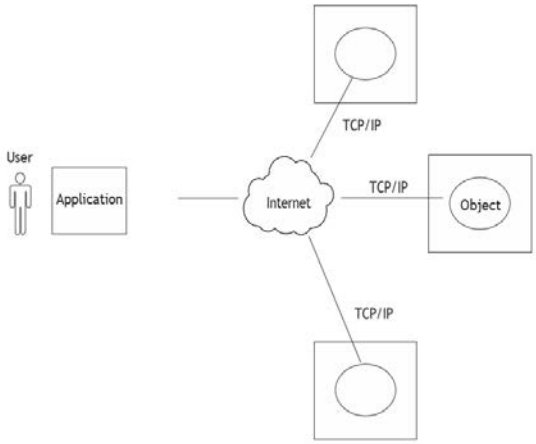
This chapter discusses the background of distributed computing and the evolution of Internet-enabled technologies by focusing on the following:

1. The definition of distributed computing
2. The importance of distributed computing
3. Core distributed computing technologies such as the following:
    - Client/server
    - CORBA
    - Java RMI
4. Common challenges in distributed computing
5. The role of J2EE and XML in distributed computing
6. Emergence of Web services and service-oriented architectures

<u>What Is Distributed Computing?</u>

In the early years of computing, mainframe-based applications were the best-fit solution for executing large-scale data processing applications. With the advent of personal computers (PCs), the concept of software programs running on standalone machines became much more popular in terms of the cost of ownership and the ease of application use. With the number of PC-based application programs running on independent machines growing, the communications between such application programs became extremely complex and added a growing challenge in the aspect of application-to-application interaction. Lately, network computing gained importance, and enabling remote procedure calls (RPCs) over a network protocol called Transmission Control Protocol/Internet Protocol (TCP/IP) turned out to be a widely accepted way for application software communication. Since then, software applications running on a variety of hardware platforms, operating systems, and different networks faced some challenges when required to communicate with each other and share data. This demanding requirement lead to the concept of distributed computing applications.

As a definition, "Distributing Computing is a type of computing in which different components and objects comprising an application can be located on different computers connected to a network" (www.webopedia.com, May 2001). Below diagram shows a distributed computing model that provides an infrastructure enabling invocations of object functions located anywhere on the network. The objects are transparent to the application and provide processing power as if they were local to the application calling them.



Today, Sun Java RMI (Remote Method Invocation), OMG CORBA (Common Object Request Broker Architecture), Microsoft DCOM (Distributed Component Object Model), and Message-Oriented Middleware (MOM) have emerged as the most common distributed computing technologies. These technologies, although different in their basic architectural design and implementation, address specific problems in their target environments. The following sections discuss the use of distributed computing and briefly describe the most popular technologies.

The Importance of Distributed Computing

The distributed computing environment provides many significant advantages compared to a traditional standalone application. The following are some of those key advantages:

Higher performance. Applications can execute in parallel and distribute the load across multiple servers.

Collaboration. Multiple applications can be connected through standard distributed computing mechanisms.

Higher reliability and availability. Applications or servers can be clustered in multiple machines.

Scalability. This can be achieved by deploying these reusable distributed components on powerful servers.

Extensibility. This can be achieved through dynamic (re)configuration of applications that are distributed across the network.

Higher productivity and lower development cycle time. By breaking up large problems into smaller ones, these individual components can be developed by smaller development teams in isolation.

Reuse. The distributed components may perform various services that can potentially be used by multiple client applications. It saves repetitive development effort and improves interoperability between components.

Reduced cost. Because this model provides a lot of reuse of once developed components that are accessible over the network, significant cost reductions can be achieved.

Distributed computing also has changed the way traditional network programming is done by providing a shareable object like semantics across networks using programming languages like Java, C, and C++. The following sections briefly discuss core distributed computing technologies such as Client/Server applications, OMG CORBA, Java RMI and MOM.

Client-Server Applications

The early years of distributed application architecture were dominated by two-tier business applications. In a two-tier architecture model, the first (upper) tier handles the presentation and business logic of the user application (client), and the second/lower tier handles the application organization and its data storage (server). This approach is commonly called client-server applications architecture. Generally, the server in a client/server

application model is a database server that is mainly responsible for the organization and retrieval of data. The application client in this model handles most of the business processing and provides the graphical user interface of the application. It is a very popular design in business applications where the user interface and business logic are tightly coupled with a database server for handling data retrieval and processing. For example, the client-server model has been widely used in enterprise resource planning (ERP), billing, and Inventory application systems where many client business applications residing in multiple desktop systems interact with a central database server.

Below figure shows an architectural model of a typical client server system in which multiple desktop-based business client applications access a central database server.

Some of the common limitations of the client-server application model are as follows:

- Complex business processing at the client side demands robust client systems.
- Security is more difficult to implement because the algorithms and logic reside on the client side making it more vulnerable to hacking.
- Increased network bandwidth is needed to accommodate many calls to the server, which can impose scalability restrictions.
- Maintenance and upgrades of client applications are extremely difficult because each client has to be maintained separately.
- Client-server architecture suits mostly database-oriented standalone applications and does not target robust reusable component- oriented applications.

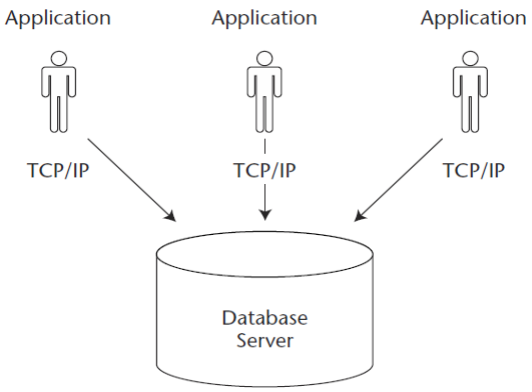


Figure 1.2 An example of a client-server application.

To avoid the limitations of normal client-server architecture Distributed Technologies came into picture.

CORBA

The Common Object Request Broker Architecture (CORBA) is an industry wide, open standard initiative, developed by the Object Management Group (OMG) for enabling distributed computing that supports a wide range of application environments. OMG is a nonprofit consortium responsible for the production and maintenance of framework specifications for distributed and interoperable object-oriented systems.

CORBA differs from the traditional client/server model because it pro- vides an object-oriented solution that does not enforce any proprietary protocols or any particular programming language, operating system, or hardware platform. By adopting CORBA, the applications can reside and run on any hardware platform located anywhere on the network, and can be written in any language that has mappings to a neutral interface definition called the Interface Definition Language (IDL). An IDL is a specific interface language designed to expose the services (methods/functions) of a CORBA remote object. CORBA also defines a collection of system-level services for

handling low-level application services like life-cycle, persistence, transaction, naming, security, and so forth. Initially, CORBA 1.1 was focused on creating component level, portable object applications without interoperability. The introduction of CORBA 2.0 added interoperability between different ORB vendors by implementing an Internet Inter-ORB Protocol (IIOP). The IIOP defines the ORB backbone, through which other ORBs can bridge and provide interoperation with its associated services.

In a CORBA-based solution, the Object Request Broker (ORB) is an object bus that provides a transparent mechanism for sending requests and receiving responses to and from objects, regardless of the environment and its location. The ORB intercepts the client's call and is responsible for find- ing its server object that implements the request, passes its parameters, invokes its method, and returns its results to the client. The ORB, as part of its implementation, provides interfaces to the CORBA services, which allows it to build custom-distributed application environments.

Figure 1.3 illustrates the architectural model of CORBA with an example representation of applications written in C, C++, and Java providing IDL bindings.

The CORBA architecture is composed of the following components:

IDL: CORBA uses IDL contracts to specify the application boundaries and to establish interfaces with its clients. The IDL provides a mechanism by which the distributed application component's interfaces, inherited classes, events, attributes, and exceptions can be specified in a standard definition language supported by the CORBA ORB.

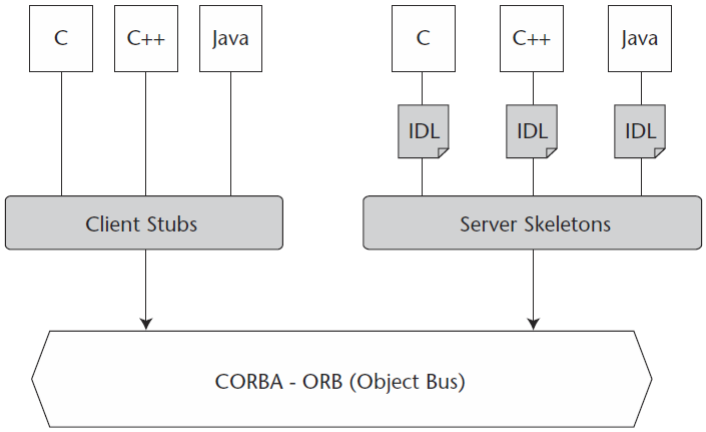


Figure 1.3 An example of the CORBA architectural model.

Java RMI

The RMI (Remote Method Invocation) is an API that provides a mechanism to create distributed application in java. The RMI allows an object to invoke methods on an object running in another JVM.

The RMI provides remote communication between the applications using two objects stub and skeleton.

Understanding stub and skeleton: RMI uses stub and skeleton object for communication with the remote object.

A remote object is an object whose method can be invoked from another JVM. Let's understand the stub and skeleton objects:

Stub

The stub is an object, acts as a gateway for the client side. All the outgoing requests are routed through it. It resides at the client side and represents the remote object. When the caller invokes method on the stub object, it does the following tasks:

- It initiates a connection with remote Virtual Machine (JVM),
- It writes and transmits (marshals) the parameters to the remote Virtual Machine (JVM),
- It waits for the result
- It reads (unmarshals) the return value or exception, and
- It finally, returns the value to the caller.

Skeleton

The skeleton is an object, acts as a gateway for the serverside object. All the incoming requests are routed through it. When the skeleton receives the incoming request, it does the following tasks:

- It reads the parameter for the remote method
- It invokes the method on the actual remote object, and
- It writes and transmits (marshals) the result to the caller.
- In the Java 2 SDK, a stub protocol was introduced that eliminates the need for skeletons.

Figure 1.4 depicts the architectural model of a Java RMI-based application solution.

The Java RMI architecture is composed of the following components:

RMI client → The RMI client, which can be a Java applet or a stand- alone application, performs the remote method invocations on a server object. It can pass arguments that are primitive data types or serializable objects.

RMI stub →The RMI stub is the client proxy generated by the rmi compiler (rmic provided along with Java developer kit—JDK) that encapsulates the network information of the server and performs the delegation of the method invocation to the server. The stub also marshals the method arguments and unmarshals the return values from the method execution.

RMI infrastructure → The RMI infrastructure consists of two layers: the remote reference layer and the transport layer. The remote reference layer separates out the specific remote reference behavior from the client stub. It handles certain reference semantics like connection retries, which are unicast/multicast of the invocation requests. The transport layer actually provides the networking infrastructure, which facilitates the actual data transfer during method invocations, the passing of formal arguments, and the return of back execution results.

RMI skeleton →The RMI skeleton, which also is generated using the RMI compiler (rmic) receives the invocation requests from the stub and processes the arguments (unmarshalling) and delegates them to the RMI server. Upon successful method execution, it marshals the return values and then passes them back to the RMI stub via the RMI infrastructure.

RMI server → The server is the Java remote object that implements the exposed interfaces and executes the client requests. It receives incoming remote method invocations from the respective skeleton, which passes the parameters after unmarshalling. Upon successful method execution, return values are sent back to the skeleton, which passes them back to the client via the RMI infrastructure.

Developing distributed applications in RMI is simpler than developing with Java sockets because there is no need to design a protocol, which is a very complex task by itself. RMI is built over TCP/IP sockets, but the added advantage is that it provides an object-oriented approach for inter- process communications. Java RMI provides the Java programmers with an efficient, transparent communication mechanism that frees them of all the application-level protocols necessary to encode and decode messages for data exchange. RMI enables distributed resource management, best processing power usage, and load balancing in a Java application model. RMI-IIOP (RMI over IIOP) is a protocol that has been developed for enabling RMI applications to interoperate with CORBA components. Although RMI had inherent advantages provided by the distributed object model of the Java platform, it also had some limitations:

- RMI is limited only to the Java platform. It does not provide language independence in its distributed model as targeted by CORBA.
- RMI-based application architectures are tightly coupled because of the connection-oriented nature. Hence, achieving high scalability in such an application model becomes a challenge.
- RMI does not provide any specific session management support. In a typical client/server implementation, the server has to maintain the session and state information of the multiple clients who access it. Maintaining such information within the server application without a standard support is a complex task.

EJB

EJB is an acronym for enterprise java bean. It is a specification provided by Sun Microsystems to develop secured, robust and scalable distributed applications.

When use Enterprise Java Bean?

- Application needs Remote Access. In other words, it is distributed.
- Application needs to be scalable. EJB applications supports load balancing, clustering and fail-over.
- Application needs encapsulated business logic. EJB application is separated from presentation and persistent layer.

Types of Enterprise Java Beans

There are 3 types of enterprise bean in java.

- **Session Bean:** Session bean contains business logic that can be invoked by local, remote or webservice client.
- **Message Driven Bean:** Like Session Bean, it contains the business logic but it is invoked by passing message.
- **Entity Bean:** It encapsulates the state that can be persisted in the database. It is deprecated. Now, it is replaced with JPA (Java Persistent API).

Difference between RMI and EJB

Both RMI and EJB, provides services to access an object running in another JVM (known as remote object) from another JVM. The differences between RMI and EJB are given below:

Common Challenges in Distributed Computing

Distributed computing technologies like CORBA, RMI, and EJB have been quite successful in integrating applications within a homogenous environment inside a local area network. As the Internet becomes a logical solution that spans and connects the boundaries of businesses, it also demands the interoperability of applications across networks. This section discusses some of the common challenges noticed in the CORBA, RMI and EJB based distributed computing solutions:

- Maintenance of various versions of stubs/skeletons in the client and server environments is extremely complex in a heterogeneous network environment.
- Quality of Service (QoS) goals like Scalability, Performance, and Availability in a distributed environment consume a major portion of the application's development time.
- Interoperability of applications implementing different protocols on heterogeneous platforms almost becomes impossible.
- Most of these protocols are designed to work well within local networks. They are not very firewall friendly or able to be accessed over the Internet.

The biggest problem with application integration with this tightly coupled approach spearheaded by CORBA, RMI, and EJB was that it influenced separate sections of the developer community who were already tied to specific platforms. Microsoft Windows platform developers used DCOM, while UNIX developers used CORBA or RMI. There was no big effort in the community to come up with common standards that focused on the interoperability between these diverse protocols, thus ignoring the importance, and hence, the real power of distributed computing.

As the distributed technologies are having some challenges to achieve interoperability everybody is looking for a technology which provides interoperability. If everybody defines their own standards then we can't achieve interoperability. To achieve interoperability all of us should follow same standards that's where WS-I (Webservices Interoperability) non-profitable organization got started.

This WS-I is governed by a Board of Directors consisting of the founding members (IBM, Microsoft, BEA Systems, SAP, Oracle, Fujitsu, Hewlett-Packard, and Intel) and two elected members (currently, Sun Microsystems and webMethods). Since joining OASIS, other organizations have joined the WS-I technical committee including CA Technologies, JumpSoft and Booz Allen Hamilton.

The organization's deliverables include profiles, sample applications that demonstrate the profiles' use, and test tools to help determine profile conformance.

According to WS-I, a profile is :- *A set of named web services specifications at specific revision levels, together with a set of implementation and interoperability guidelines recommending how the specifications may be used to develop interoperable web services.*

WS-I Org Given 4 Versions of Web Services specifications. They are

1) Basic Profile 1.0 (B.P 1.0)

2) Basic Profile 1.1 (B.P 1.1)

3) Basic Profile 1.2 (B.P 1.2)

4) Basic Profile 2.0 (B.P 2.0)

- ***B.P 1.0*** → *Version 1.0 of this profile was released in early 2004.*
- ***B.P 1.1*** → *Version 1.1 published in 2006 does not have the same scope as version 1.0. The part of version 1.0 dealing with serialization of envelopes and their representation in messages has been moved to a new profile called Simple Soap Binding Profile (SSBP)*
- ***B.P 1.2*** → *Version 1.2 was finalized in November 2010. The main new features are the support for MTOM binary attachments and WS-Addressing*
- ***B.P 2.0*** → *Version 2.0 was also published in November 2010. It uses SOAP 1.2, UDDI 3 and WS-Addressing*

As Every player in the market started providing support for Basic Profile Specifications given by WS-I, Sun also added its support for Basic Profile specifications and adopted those specifications by providing below APIs.

Java applications don't know what is SOAP protocol and how to put xml into SOAP xml. Java must provide one api to work with SOAP. That's where SUN provided SAAJ (soap with attachments api for java).

Java applications don't know how to generate or read WSDL. Java must provide one api to work with SOAP. That's where SUN provided WSDL4J api.

Java applications must connect with UDDI registry to publish/discover WSDL documents. That's where JAX-R (Java API for XML Registries) api provided by sun to work with UDDI registries.

Keeping all these APIs together, sun provided JAX-RPC AND JAX-WS APIS.

SOAP Webservices			
S.No	**Specification**	**API**	**Implementations**
1	B.P 1.0	JAX-RPC	SI (Sun Implementation)
			APACHE AXIS Implementaton
			Oracle Weblogic Webservices
			IBM Websphere Webservices
2	B.P 1.1	JAX-WS	RI (Reference Implementation)
			METRO Implementation
			APACHE AXIS2 Implementation
			APACHE CXF Implementation
			Oracle Weblogic Webservices
			IBM Websphere Webservices

Web Services Architecture

Web services are based on the concept of service-oriented architecture (SOA). SOA is the latest evolution of distributed computing, which enables software components, including application functions, objects, and processes from different systems, to be exposed as services.

According to Gartner research (June 15, 2001), "Web services are loosely coupled software components delivered over Internet standard technologies."

In short, Web services are self-describing and modular business applications that expose the business logic as services over the Internet through programmable interfaces and using Internet protocols for the purpose of providing ways to find, subscribe, and invoke those services.

Based on XML standards, Web services can be developed as loosely coupled application components using any programming language, any protocol, or any platform. This facilitates delivering business applications as a service accessible to anyone, anytime, at any location, and using any platform.

Basic Operational Model of Web Services

The Web Services architecture is based upon the interactions between three roles: service provider, service registry and service requestor. The interactions involve the publish, find and bind operations. Together, these roles and operations act upon the Web Services artifacts: the Web service software module and its description. In a typical scenario, a service provider hosts a network-accessible software module (an implementation of a Web service). The service provider defines a service description for the Web service and publishes it to a service

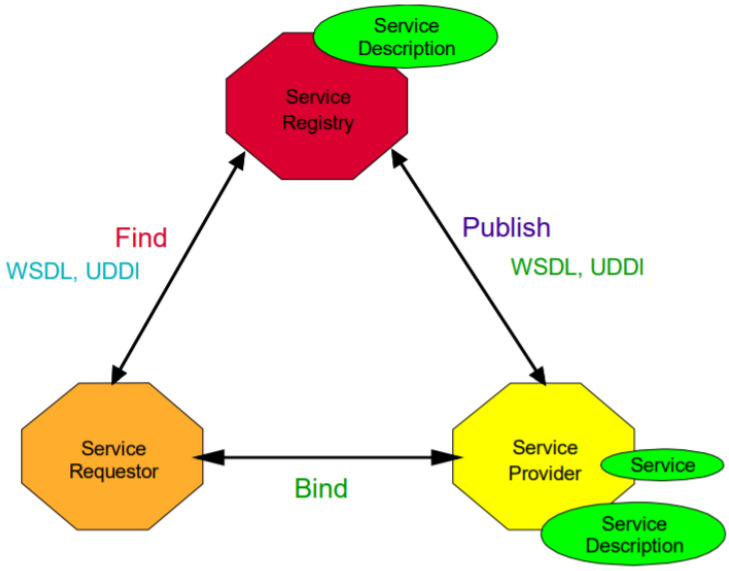
requestor or service registry. The service requestor uses a find operation to retrieve the service description locally or from the service registry and uses the service description to bind with the service provider and invoke or interact with the Web service implementation. Service provider and service requestor roles are logical constructs and a service can exhibit characteristics of both. Below image illustrates these operations, the components providing them and their interactions.

These roles and relationships are defined as follows:

Service Provider: From a business perspective, this is the business that requires Service requestor. certain functions to be satisfied. From an architectural perspective, this is the application that is looking for and invoking or initiating an interaction with a service. The service requestor role can be played by a browser driven by a person or a program without a user interface, for example another Web service.

Service Requestor: From a business perspective, this is the business that requires Service requestor. certain functions to be satisfied. From an architectural perspective, this is the application that is looking for and invoking or initiating an interaction with a service. The service requestor role can be played by a browser driven by a person or a program without a user interface, for example another Web service.

Service Registry: Service registry. This is a searchable registry of service descriptions where service providers publish their service descriptions. Service requestors find services and obtain binding information (in the service descriptions) for services during development for static binding or during execution for dynamic binding. For statically bound service requestors, the service registry is an optional role in the architecture, because a service provider can send the description directly to service requestors. Likewise, service requestors can obtain a service description from other sources besides a service registry, such as a local file, FTP site, Web site, Advertisement and Discovery of Services (ADS) or Discovery of Web Services (DISCO).



Operations in a Web Service Architecture

For an application to take advantage of Web Services, three behaviors must take place: publication of service descriptions, lookup or finding of service descriptions, and binding or invoking of services based on the service description. These behaviors can occur singly or iteratively. In detail, these operations are:

Publish → To be accessible, a service description needs to be published so that the Publish. service requestor can find it. Where it is published can vary depending upon the requirements of the application (see "Service Publication" for more details).

Find → In the find operation, the service requestor retrieves a service description directly Find. or queries the service registry for the type of service required (see "Service Discovery" for more details). The find operation can be involved in two different lifecycle phases for the service requestor: at design time to retrieve the service's interface description for program development, and at runtime to retrieve the service's binding and location description for invocation.

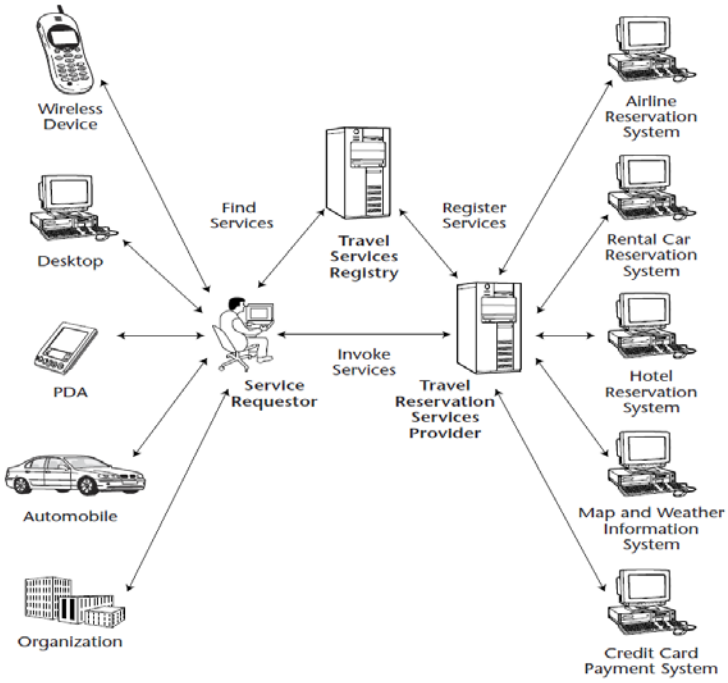
Bind → Eventually, a service needs to be invoked. In the bind operation the service Bind. requestor invokes or initiates an interaction with the service at runtime using the binding details in the service description to locate, contact and invoke the service.

Artifacts of a Web Service

Service → Where a Web service is an interface described by a service description, its Service. implementation is the service. A service is a software module deployed on network accessible platforms provided by the service provider. It exists to be invoked by or to interact with a service requestor. It can also function as a requestor, using other Web Services in its implementation.

Service Description → The service description contains the details of the interface and Service Description. implementation of the service. This includes its data types, operations, binding information and network location. It could also include categorization and other metadata to facilitate discovery and utilization by service requestors. The service description might be published to a service requestor or to a service registry.

Consider the simple example shown below where a travel reservation services provider exposes its business applications as Web services supporting a variety of customers and application clients. These business applications are provided by different travel organizations residing at different networks and geographical locations.



The following is a typical scenario:

1. The Travel service provider deploys its Web services by exposing the business applications obtained from different travel businesses like airlines, car-rental, hotel accommodation, credit card payment, and so forth.

2. The service provider registers its business services with descriptions using a public or private registry. The registry stores the information about the services exposed by the service provider.

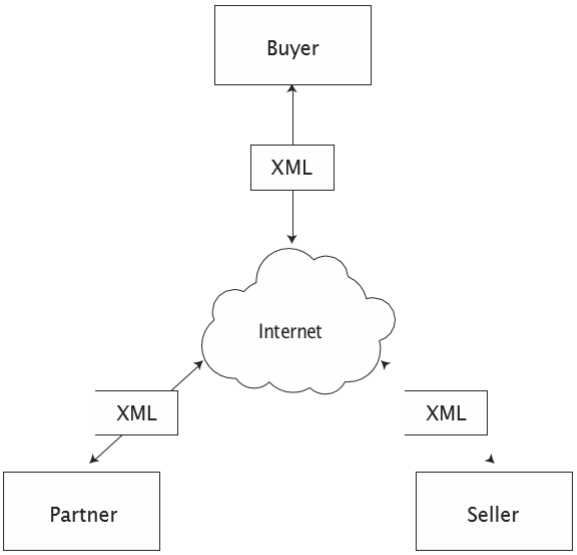
3. The customer discovers the Web services using a search engine or by locating it directly from the registry and then invokes the Web services for performing travel reservations and other functions over the Internet using any platform or device.

4. In the case of large-scale organizations, the business applications consume these Web services for providing travel services to their own employees through the corporate intranet.

The previous example provides a simple scenario of how an organization's business functionalities can be exposed as Web services and invoked by its customers using a wide range of application clients.

Motivation and Characteristics

Web-based B2B communication has been around for quite some time. These Web-based B2B solutions are usually based on custom and proprietary technologies and are meant for exchanging data and doing transactions over the Web. However, B2B has its own challenges. For example, in B2B communication, connecting new or existing applications and adding new business partners have always been a challenge. Due to this fact, in some cases the scalability of the underlying business applications is affected. Ideally, the business applications and information from a partner organization should be able to interact with the application of the potential partners seamlessly without redefining the system or its resources. To meet these challenges, it is clearly evident that there is a need for standard protocols and data formatting for enabling seamless and scalable B2B applications and services. Web services provide the solution to resolve these issues by adopting open standards. Figure 2.2 shows a typical B2B infrastructure (e-marketplace) using XML for encoding data between applications across the Internet.



Using XML for encoding data in a B2B communication.

Web services enable businesses to communicate, collaborate, and conduct business transactions using a lightweight infrastructure by adopting an XML-based data exchange format and industry standard delivery protocols.

The basic characteristics of a Web services application model are as follows:

- Web services are based on XML messaging, which means that the data exchanged between the Web service provider and the user are defined in XML.
- Web services provide a cross-platform integration of business applications over the Internet.
- To build Web services, developers can use any common programming language, such as Java, C, C++, Perl, Python, C#, and/or Visual Basic, and its existing application components.
- Web services are not meant for handling presentations like HTML context—it is developed to generate XML for uniform accessibility through any software application, any platform, or device.
- Because Web services are based on loosely coupled application components, each component is exposed as a service with its unique functionality.
- Web services use industry-standard protocols like HTTP, and they can be easily accessible through corporate firewalls.
- Web services can be used by many types of clients.
- Web services vary in functionality from a simple request to a complex business transaction involving multiple resources.
- All platforms including J2EE, CORBA, and Microsoft .NET provide extensive support for creating and deploying Web services.
- Web services are dynamically located and invoked from public and private registries based on industry standards such as UDDI and ebXML.

Why to Use Web Services?

Traditionally, Web applications enable interaction between an end user and a Web site, while Web services are service-oriented and enable application- to-application communication over the Internet and easy accessibility to heterogeneous applications and devices. The following are the major technical reasons for choosing Web services over Web applications:

- Web services can be invoked through XML-based RPC mechanisms across firewalls.
- Web services provide a cross-platform, cross-language solution based on XML messaging.

- Web services facilitate ease of application integration using a light- weight infrastructure without affecting scalability.
- Web services enable interoperability among heterogeneous applications.

Core Web Services Standards

The five core Web services standards and technologies for building and enabling Web services are XML, SOAP, WSDL and UDDI. An overview of each is presented in the following sections.

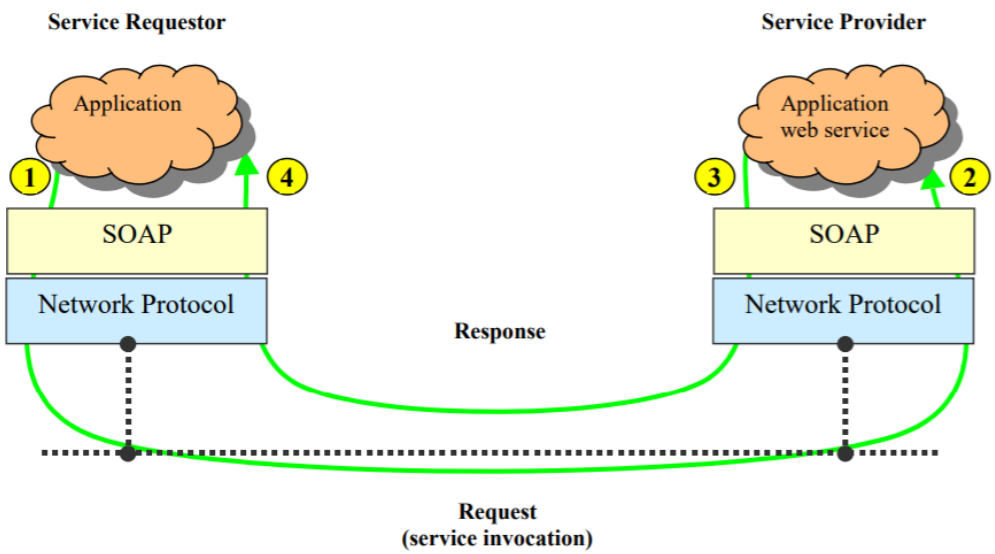
Extensible Markup Language (XML)

In February 1998, the Worldwide Web Consortium (W3C) officially endorsed the Extensible Markup Language (XML) as a standard data for- mat. XML uses Unicode, and it is structured self-describing neutral data that can be stored as a simple text document for representing complex data and to make it readable. Today, XML is the de facto standard for structuring data, content, and data format for electronic documents. It has already been widely accepted as the universal language lingua franca for exchanging information between applications, systems, and devices across the Internet. In the core of the Web services model, XML plays a vital role as the com- mon wire format in all forms of communication. XML also is the basis for other Web services standards. By learning XML, you will be well prepared to understand and explore Web services. For more information on XML, go to Chapter 8, "XML Processing and Data Binding with Java APIs," or to the official W3C Web site for XML at www.w3c.org/XML/.

Simple Object Access Protocol (SOAP)

Simple Object Access Protocol, or SOAP, is a standard for a lightweight XML-based messaging protocol. It enables an exchange of information between two or more peers and enables them to communicate with each other in a decentralized, distributed application environment. Like XML, SOAP also is independent of the application object model, language, and running platforms or devices. SOAP is endorsed by W3C and key industry vendors like Sun Microsystems, IBM, HP, SAP, Oracle, and Microsoft. These vendors have already announced their support by participating in the W3C XML protocol-working group. The ebXML initiative from UN/CEFACT also has announced its support for SOAP.

SOAP consists of three parts: an envelope that defines a framework for describing what is in a message, a set of encoding rules for expressing instances of application-defined data types, and a convention for representing remote procedure calls (RPCs) and responses. SOAP can be used in combination with or re-enveloped by a variety of network protocols such as HTTP, SMTP, FTP, RMI over IIOP or MQ.



As per above diagram

- At (1) a service requestor's application creates a SOAP message. This SOAP message is the request that invokes the Web service operation provided by the service provider. The XML document in the body of the message can be a SOAP RPC request or a document-centric message as indicated in the service description. The service requestor presents this message together with the network address of the service provider to the SOAP infrastructure (for example, a SOAP client runtime). The SOAP client runtime interacts with an underlying network protocol (for example HTTP) to send the SOAP message out over the network.

- At (2) the network infrastructure delivers the message to the service provider's SOAP runtime (for example a SOAP server). The SOAP server routes the request message to the service provider's Web service. The SOAP runtime is responsible for converting the XML message into programming language-specific objects if required by the application. This conversion is governed by the encoding schemes found within the message.

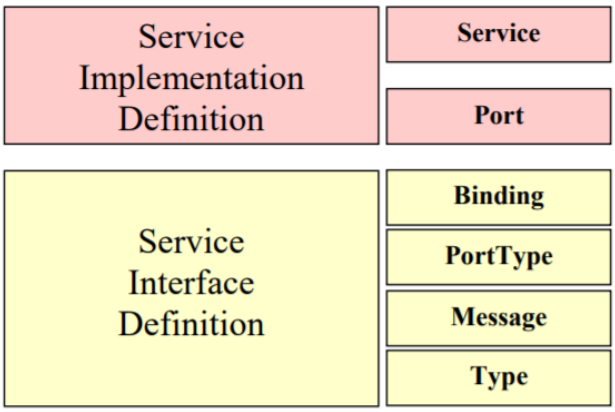
- The Web service is responsible for processing the request message and formulating a response. The response is also a SOAP message. At (3) the response SOAP message is presented to the SOAP runtime with the service requestor as the destination. In the case of synchronous request/response over HTTP, the underlying request/response nature of the networking protocol is used to implement the request/response nature of the messaging. The SOAP runtime sends the SOAP message response to the service requestor over the network.

- At (4) the response message is received by the networking infrastructure on the service requestor's node. The message is routed through the SOAP infrastructure; potentially converting the XML message into objects in a target programming language. The response message is then presented to the application.

Web Services Definition Language (WSDL)

The Web Services Description Language (WSDL) is an XML-based interface definition language that is used for describing the functionality offered by a web service. The acronym is also used for any specific WSDL description of a web service (also referred to as a WSDL file), which provides a machine-readable description of how the service can be called, what parameters it expects, and what data structures it returns. Therefore, its purpose is roughly similar to that of a type signature in a programming language.

The current version of WSDL is WSDL 2.0. The meaning of the acronym has changed from version 1.1 where the "D" stood for "Definition".



Types → a container for data type definitions using some type system (such as XSD).

Message → an abstract, typed definition of the data being communicated.

Operation → an abstract description of an action supported by the service.

Port Type → an abstract set of operations supported by one or more endpoints.

Binding → a concrete protocol and data format specification for a particular port type.

Port → a single endpoint defined as a combination of a binding and a network address.

Service → a collection of related endpoints.

WSDL 1.1 Term	WSDL 2.0 Term	Description
Service	Service	Contains a set of system functions that have been exposed to the Web-based protocols.
Port	Endpoint	Defines the address or connection point to a Web service. It is typically represented by a simple HTTP URL string.
Binding	Binding	Specifies the interface and defines the SOAP binding style (RPC/Document) and transport (SOAP Protocol). The binding section also defines the operations.
PortType	Interface	Defines a Web service, the operations that can be performed, and the messages that are used to perform the operation.
Operation	Operation	Defines the SOAP actions and the way the message is encoded, for example, "literal." An operation is like a method or function call in a traditional programming language.
Message	N/A	Typically, a message corresponds to an operation. The message contains the information needed to perform the operation. Each message is made up of one or more logical parts. Each part is associated with a message-typing attribute. The message name attribute provides a unique name among all messages. The part name attribute provides a unique name among all the parts of the enclosing message. Parts are a description of the logical content of a message. In RPC binding, a binding may reference the name of a part in order to specify binding-specific information about the part. A part may represent a parameter in the message; the bindings define the actual meaning of the part. Messages were removed in WSDL 2.0, in which XML schema types for defining bodies of inputs, outputs and faults are referred to simply and directly.
Types	Types	Describes the data. The XML Schema language (also known as XSD) is used (inline or referenced) for this purpose.

Universal Description, Discovery, and Integration (UDDI)

Universal Description, Discovery, and Integration, or UDDI, defines the standard interfaces and mechanisms for registries intended for publishing and storing descriptions of network services in terms of XML messages. It is like the yellow pages or a telephone directory where businesses list their products and services. Web services brokers use UDDI as a standard for registering the Web service providers. By communicating with the UDDI registries, the service requestors locate services and then invoke them.

UDDI was included in the Web Services Interoperability (WS-I) standard as a central pillar of web services infrastructure, and the UDDI specifications supported a publicly accessible Universal Business Registry in which a naming system was built around the UDDI-driven service broker.

UDDI has not been as widely adopted as its designers had hoped. IBM, Microsoft, and SAP announced they were closing their public UDDI nodes in January 2006. The group defining UDDI, the OASIS Universal Description, Discovery, and Integration (UDDI) Specification Technical Committee voted to complete its work in late 2007 and has been closed. In September 2010, Microsoft announced they were removing UDDI services from future versions of the Windows Server operating system. Instead, this capability would be moved to BizTalk Server.In 2013, Microsoft further announced the deprecation of UDDI Services in BizTalk Server.

A UDDI business registration consists of three components:

- White Pages — address, contact, and known identifiers;
- Yellow Pages — industrial categorizations based on standard taxonomies;
- Green Pages — technical information about services exposed by the business.

XML

Introduction

- XMl stands for extensible markup language
- Unlike C, C++ and Java etc. Xml is not a programming language. It is a markup language
- XMl was designed for transport and store data
- XML was defined and governed by W3C Org
- The first and final version of XML is XML 1.0

XML is not a database: You're not going to replace an Oracle or MySQL server with XML. A database can contain XML data. Either as a VARCHAR or a BLOB or as some custom XML data type, but the database It self is not an XML document. You can store XML data in database on a server or retrieve data from a database in XML format.

What is mean by Extensible: Extensible means that the language can be extended and adapted to meet many different needs. i.e XML allows developers and writers to invent the elements they need as they need them.

There is some other markup language also there in the market that is HTML

HTML was designed to display data, with focus on how data looks.

Differences between HTML and XML

XML	HTML
1) Extensible markup language	1) Hypertext markup language
2) It is used for transport and store the data	2) It is used to display data
3) All the Elements are user-defined	3) All the elements in HTML are predefined
4) It has infinite set of tags	4) It has finite set of tags
5) XML documents are re-usable	5) HTML does not support for re-usability
6) Every start tag must have end tag	6) Every start tag need not have end tag
7) XML is DYNAMIC (i.e. Content is changed every time page is loaded)	7) HTML is static(i.e. Content does not changed every time page is loaded)
8) XML attribute value must be enclosed with quote(")	8) HTML attribute value can be present without quotation mark
9) It is case-sensitive	9) It is not case sensitive

Every language has keywords, If you take example as C, it has keywords like (if, for, while, do, break, continue etc.) but when it comes to XML there are no keywords or reserved words. Whatever you write will become the element of that XML document.

XML Declaration

Every XML document must start with **PROLOG**: - prolog stands for processing instruction, and typically used for understanding about the version of XML used and the data encoding used.

***Example*:** - <? xml version="1.0" encoding="UTF-8" standalone="yes"?>

<? Xml: XML parser uses the first five characters (<? xml) to make some reasonable guess about the encoding such as whether the document uses single-byte or multi-byte character set.

Version: The attribute should have all the value 1.0. Under very unusual circumstances, it may also have the value 1.1. Since specifying version="1.1" limits the document to the most recent versions of only a couple of

parsers, and since all Xml 1.1 parsers must also support XML 1.0, you don't want to casually set the version 1.1.

Encoding: XML documents are assumed to be encoded in the UTF-8 variable length encoding of the Unicode character set. This is strict superset of ASCII, so pure ASCII text files are also UTF-8 documents.

Standalone: Standalone attribute accepts only 2 values i.e. 1) no 2) yes

If standalone="no": - Application may be required to read an external DTD

If standalone=" yes" :- Application need to read internal DTD

XMl Element: The data between angular brackets (i.e. "< >") is called Element

Ex: employee

Tag: The element with includes angle brackets is called Tag.

Ex: <employee>

In XML there are two types of elements as follows

Start Element/Opening Tag: - Start element is the element which is written in <elementname> indicating the start of a block.

End Element/End Tag: - End element is the element which is written in </elementname> indicating the end of a block.

An Element can contain:

1. Other elements (child elements)
2. Text-data
3. Attributes
4. Mix of all

XML elements naming rules

- Elements names are case-sensitive
- Element name must start with letter or underscore
- Element name cannot start with letter xml (Or XML, or Xml etc.)
- Element name cannot contain spaces
- Any name can be used, no word are reserved except XML.

XML Attribute

If we want to have additional information attach to an element, instead of having it as content or another element, we can write it as an Attribute of the element.

Example:

```
<employee type= "permanent">
        <id> 1001 </id>
</employee>
```

In the above example "employee" is an element which contains one attributes type which acts as a supplementary information, <id> is the sub-element of the <employee> element.

Note: Attributes must be enclosed with single quote or double quotes

<employee id="101" name="Ashok" /> (Valid)

<employee id="101" name=Ashok /> (In-valid)

XML Comments

XML comments are syntactically similar to HTML comments

Syntax: <! - - This is the comment- - >

Predefined Entities

Some characters have meaning in XML

For Example: If you place a character like "<" inside XML element, it will generate error because the parser interprets it as the start of new element.

The following code will generate an XML error

<salary> if salary <15000 then </salary>

To avoid above error replace "<" character with entity reference

<salary> if salary < 15000 then </salary>

Less than (<)	:	<
Greater than (>)	:	>
Ampersand (&)	:	&
Apostrophe (')	:	'
Quotation (")	:	"

XML Well-formness

As how any programing language has syntax in writing its code, Well-formness of XML document talks about how to write an XML document. Well-formness indicates the readability nature of an XML document. In other way if an XML document is said to be well-formed then it is readable in nature.
Following are the rules that describe the Well-formness of an XML Document.

* Every XML document must start with **PROLOG**: - prolog stands for processing
* Every start tag must have end tag
* Elements may nest but may not overlap
* There must be exactly one root element
* Attribute value must be quoted
* An element cannot have two attributes with same name

Well-formed XML	Not Well-formed XML
<employee> <id>1208</id> <name>Ashok</name> </employee>	<employee type=permanent> <id>1208</id> <name>Ashok</name> </employee>
Reason: It follows all the XML rules	**Reason: attribute values is not enclosed with "**

Not Well-formed XML	Not Well-formed XML
<employee> <id>1234</id> <name>Ashok</name> </employee> <employee> <id>2456</id> <name>Ashok</name> <employee>	<employee> <id>1234 <name>Ashok<name> </employee>
Reason: Only one root element is allowed	**Reason: Tags are not ended properly**

For checking whether the XML document is well-formed or not some of the editors are there. They are

1) XML copy editor
2) Edit XML Editor
3) Altova XM Spy
4) Oxygen XML Editor
5) XML Writer XML Editor
6) Stylus Studio
7) Liquid XML Studio
8) WMHelp XML pad
9) XMLLmind XML Editor
10) Notepad++

XML Validation

Every XML in-order to parse (read) should be well-formed in nature. As said earlier well-formness of an XML document indicates whether it is readable or not, it doesn't talk about whether the data contained in it is valid or not.

Validity of the XML document would be defined by the application which is going to process your XML document. Let's consider a scenario as follows.

```
<?xml version="1.0" encoding="UTF-8"?>
<purchaseOrder>
        <orderItems>
                <item>
                    <ItemCode>IC201</ItemCode>
                    <quantity>xyz</quantity>
                </item>
        </orderItems>
        <shipping-address>
                <addrLine1>S.RNagar</addrLine1>
                <addrLine2>CDAC</addrLine2>
                <city>hyd</city>
                <state>TG</state>
                <zip>500031</zip>
                <country>India</country>
        </shipping-address>
</purchaseOrder>
```

purchase-order.xml

In the above purchase-order.xml even though it confirms to all the well-formness rules, it cannot be used for business transaction, as the <quantity> element carries the data as “xyz” which doesn’t makes any sense.

So, in order to check for data validity, we need to define the validation criteria of an XML document in either a DTD or XSD document.

DTD

Introduction

- DTD stands for Document Type Definition.
- It is the document which defines the structure and the legal elements and attributes of an XML document
- An application can use a DTD to verify that XML data is valid
- In a language before using a variable we need to declare it. Similarly, before using an element in XML first we need to declare it in DTD.
- DTD files will be saved with .dtd extension

Before start writing DTD, first we need to know two kinds of elements. They are

1) **Simple Elements**
2) **Compound Elements**

Let us take a sample po.xml file to demonstrate these elements

```
<?xml version="1.0" encoding="UTF-8"?>
<purchaseOrder>
        <orderItems>
                <item>
                   <ItemCode>IC201</ItemCode>
                   <quantity>xyz</quantity>
                </item>
        </orderItems>
        <shipping-address>
                <addrLine1>S.RNagar</addrLine1>
                <addrLine2>CDAC</addrLine2>
                <city>hyd</city>
                <state>TG</state>
                <zip>500031</zip>
                <country>India</country>
        </shipping-address>
</purchaseOrder>
```

purchase-order.xml

Simple Elements: Elements which carry data are called simple elements. Simple element can contain only value (data). It may not contain any child.

Syntax: <!ELEMENT elementName (CONTENT-TYPE OR CONTENT MODEL)>
For e.g. <!ELEMENT itemCode (#PCDATA)>

In the above XML file, simple Type elements are: itemCode, quantity, addrLine1, addrLine2, city, state, zip and county.

Compound Elements: An element which contains sub-elements under it is called compound element.

Syntax: <! ELEMENTelementName(sub-elem1,sub-elem2…)

For e.g.

```
<!ELEMENTitemCode(#PCDATA)>
<!ELEMENTquantity(#PCDATA)>
<!ELEMENTitems(itemCode,quantity)>
```

For above xml, DTD looks like shown below

Occurrences of an Element under another element

In the above xml if we observe the <items> element, it can contain any number of <orderItems> elements in it, but at least one item element must be there for a < purchaseOrder>. This is called occurrence of an element under another element, to indicate this we use three symbols.

? – Represents the sub element under a parent element can appear zero or one- time (0/1).
+ - indicates the sub element must appear at least once and can repeat any number of times (1 – N)
***** - indicates the sub element is optional and can repeat any number of times (0 - N)
You will mark the occurrence of an element under another element as follows.

<! Element elementname (sub-elem1 (? /+/*), sub-elem2 (? /+/*)>

Leaving any element without any symbol indicates it is mandatory and at max can repeat only once.

Elements with any contents

Elements declared with the content type as ANY, can contain any combination of parsable data.

<! ELEMENT elementname ANY>
<! ELEMENT mailBody ANY>
In an e-mail body part we have content which can be mixture of any parsable characters which can be declared as ANY type.

Elements with either/or content

Let's consider an example where in an email the following elements will be there to, from, subject and mailBody. In these elements to and from or mandatory elements and either subject or mailBody should be present but not both. To declare the same we need to use or content separator instead of sequence separator.

<! ELEMENT mail (to, from, (subject | mailBody)>

For the above declaration the xml looks as below.

```
<mail>
  <to>toaddr.com</to>
  <from>fromaddr.com<from>
  <subject>mysub</subject>
</mail>
```

(OR)

```
<mail>
  <to>toaddr.com</to>
  <from>fromaddr.com<from>
  <mailBody>mysub</mailBody>
</mail>
```

Declaring Mixed content

We can even declare an element with mixture of parable data and elements called mixed content as follows.
<! ELEMENT mail (#PCDATA| to| from| subject| mailBody)*>

Declaring attribute for an element

Attributes provide extra information about elements

Syntax: - <! ATTLIST elementName attributeName attributeType attributeValue>

As shown above to declare an attribute you need to use the tag ATTLIST and elementname stands for which element you want to declare the attribute and attributeName stands for what is the attribute you want to have in that element.

The attributeType can be the following:

Type	Description
CDATA	The value is Character data
(en1|en2|..)	The value must be one from the enumerated list of values.
ID	The value is unique id.
IDREF	The value is the id of another element
NMTOKEN	The value is a valid XML element name

The attributeValue can be the following:

Value	Description
#REQUIRED	The attribute is required
#IMPLIED	The attribute is not required
#FIXED value	The attribute value is fixed.

Default Attribute Value

<! ELEMENT shippingAddress (addressLine1, addressLine2, city, state, zip, country)>
<! ATTLIST shippingAddress type CDATA "permanent" >

Inxml :- <shippingAddress type="permanent">.... </shippingAddress>

#REQUIRED

<! ATTLIST shippingAddress type CDATA #REQUIRED> this indicates type attribute is mandatory in shipping address element.

#IMPLIED

<! ATTLIST shippingAddress type CDATA #IMPLIED>, this indicates the type attribute in shippingAddress element is optional.

#FIXED

<! ATTLIST shippingAddress type CDATA #FIXED "permanent">, this indicates the type attribute in shippingAddress element must contain only the value as permanent.

Enumerated Attribute Values

<! ATTLIST shippingAddresstype (permanent | temporary) "permanent" >

This indicates the type attribute in shippingAddress element should contain only two possible values either permanent or temporary.

DTDs are divided into 2 types

1. **Internal DTD**
2. **External DTD**

Internal DTD

If the DTD is declared inside the XML file, it should be wrapped in a DOCTYPE definition with the following syntax:

<!DOCTYPE root-element [element-declarations] >

Below is the sample XML with Internal DTD

```
<?xml version="1.0"?>
<!DOCTYPE bookstore [
  <!ELEMENT bookstore (name,topic+)>
  <!ELEMENT topic (name,book*)>
  <!ELEMENT name (#PCDATA)>
  <!ELEMENT book (title,author)>
  <!ELEMENT title (#CDATA)>
  <!ELEMENT author (#CDATA)>
  <!ELEMENT isbn (#PCDATA)>
  <!ATTLIST book isbn CDATA "0">
  ]>
<bookstore>
  <name>Mike's Store</name>
  <topic>
    <name>XML</name>
    <book isbn="123-456-789">
      <title>Mike's Guide To DTD's and XML Schemas<</title>
      <author>Mike Jervis</author>
    </book>
  </topic>
</bookstore>
```

DTD

External DTD

External DTD are shared between multiple XML documents. Any changes are update in DTD document effect or updated come to all XML documents.

External DTD two type:

a. **Private DTD**
b. **Public DTD**

Private DTD Private DTD identify by the SYSTEM keyword. Access for single or group of users.

You can specify rules in external DTD file .dtd extension. Later in XML file <! DOCTYPE ... > declaration to linking this DTD file.

```
<!DOCTYPE root_element SYSTEM  "dtd_file_location" >
```

Public DTD Public DTD identify by the PUBLIC keyword. Access any users and our XML editor are know the DTD.

```
<!DOCTYPE root_element  PUBLIC  "dtd_name"  "dtd_file_location">
```

Below XML is linked with Public DTD

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE purchaseOrder PUBLIC  "C:\Ashok\Webservices\DTD\po.dtd">
<purchaseOrder>
        <orderItems>
                <item>
                        <ItemCode>IC202</ItemCode>
                        <quantity>20</quantity>
                </item>
        </orderItems>
        <shipping-address>
                <addrLine1>S.RNagar</addrLine1>
                <addrLine2>CDAC</addrLine2>
                <city>hyd</city>
                <state>TG</state>
                <zip>50031</zip>
                <country>India</country>
        </shipping-address>
</purchaseOrder>
```

Once DTD is linked with XML then we can validate our XML against DTD (see in below image)

Drawbacks with DTD

- It does not support the namespaces. Namespace is a mechanism by which element and attribute names can be assigned to groups. However, in a DTD namespace should be defined within the DTD, which violates the purpose of using namespaces.
- It supports only the text string data type.
- It is not object oriented. Hence, the concept of inheritance cannot be applied on the DTDs.
- Limited possibilities to express the cardinality for elements.

To overcome the above drawbacks of DTD we will use XSD to define structure of the XML.

XSD

Introduction

XML Schema is an XML-based alternative to DTDs. An XML Schema describes the structure of an XML document. The XML Schema language is also referred to as XML Schema Definition (XSD). The purpose of an XML Schema is to define the legal building blocks of an XML document, just like a DTD.

- XSD stands for XML Schema Definition
- XSD owned by world wide web consortium (W3Org)
- In 2001, the W3C developed a new schema language to address many of the shortcomings of DTD
- XSD schema is used to define the structure of an XML document
- The main goal of XSD is to validate the XML document whether it is valid or not
- XSD is also an xml (It is special XML with fixed structure and fixed elements)
- XSD is more powerful and type strict in nature
- Schemas are another approach used by the XML parser to validate an XML document
- It is text file with .xsd extension

An XML Schema defines

- **Elements that can appear in a document**
- **Defines attributes that can appear in a document**
- **Defines which elements are child elements**
- **Defines the order of child elements**
- **Defines the number of child elements**
- **Defines whether an element is empty or can**
- **Defines data types for elements and attributes**
- **Defines default and fixed values for elements and attributes**

XML Schemas are richer and powerful than DTD because:

1. XML Schemas are **extensible** to future additions
2. XML Schemas themselves are **written in XML**
3. XML Schemas support **data types**
4. XML Schemas support **namespaces**

XML Schemas are Extensible

XML Schemas are extensible, because they are written in XML. With an **Extensible Schema** definition we can:

1. Reuse one Schema in other Schemas
2. Create our own data types derived from the standard types
3. Reference multiple schemas in the same document

Data types

One of the **greatest strength** of XML Schemas is the **support for data types**. By using data types, it is easier to:

1. Describe allowable document content
2. Validate the correctness of data
3. Work with data from a database
4. Define data facets (restrictions on data)
5. Define data patterns (data formats)

As XSD is also an xml, it will start with a prolog and it also contains only one root element. The <schema> element is the root element of every XML Schema.

The <schema> element contains following attributes:

Attribute name	Description	Example
xmlns:xs	Elements and Data types from "http://www.w3.org/2001/XMLSchema" namespace.	xmlns:xs=" http://www.w3.org/2001/XMLSchema"
targetNamespace	User defined elements and data types in our schema belongs to our namespace such as "www.ashoksit.com"	targetNamespace="www.ashoksoft.com">
Xmlns	Indicates default name space.	Xmlns="www.ashoksoft.com"
elementFromDefault	Elements must be namespace qualified.	elementFromDefault="qualified"

The above attributes we discuss in detail in XSD namespaces concept.

As we have already discussed in DTD, an XML contains 2 types of elements. They are

1) Simple Elements
2) Compound Elements

In Java, we have to store some value into a variable, first we need to declare the variable similarly if we want to write one element in XML first we should declare that element in DTD or XSD.

Declaring Simple Elements in XSD

```
<element name="elementName" type="dataType" />
```

Name – name attributes holds element name

Type - type attribute holds element data type

XSD	<element name="zipcode" type="int" />
XML	<zipcode>500081</zipcode>

Declaring Compound Elements in XSD

If we want to some value into a variable first we must declare a variable with datatype (Based on nature of the data we want to store into that variable we will decide data type for that variable).

If we want to store a person name into variable then we will choose String data type like below

String personName;

If we want to store person age then we will declare a variable with int datatype like below

Int personage;

Similarly, I want to store student details like rollNumber, name and rank in a variable called 's'. Do we have any predefined datatype in Java to store student values (rollNumber, name and rank) like this? – We don't have any such datatype in java.

As we don't have any predefined data type, we must create our own datatype for 's' variable to store student values. We can consider Class as a user-defined datatype so we will create a Student class to store student details and we can use that Student class as a data type for variable 's' like below

```
public   class   Student {

     int  rollNumber;
    String name;
    int rank;

}
```

```
Student  s ;  // declaration

s = new Student ( ) ;  //instantiation

//storing values into s variable
s.rollNumber = 101;
s.name = "Ashok" ;
s.rank = 1 ;
```

If we want to write one element in XML first we should declare in XSD. I want to write one element <order-item> in xml like below

```
<order-item>
  <item-code> IC001 </item-code>
  <quantity> 10 </quantity>
</order-item>
```

Note: As we can see <order-item> contains child elements this is called Compound element.

To write <order-item> element in XML first we should declare this element in XSD with data type.

What datatype we can use for <order-item> element? – It's based on nature of the data we want to store.

In <order-item> I want to store <item-code> and <quantity> elements. Do we have any datatype in XSD to store <item-code> and <quantity> elements? – No.

As discussed above to store Student data we have don't have predefined data type hence we have created Student class as user-defined datatype. Similarly, as we don't have any predefined datatype for <order-item> we have to create user-defined datatype in XSD. To create User-defined datatypes in XSD we will use **ComplexType**.

A complex type element is an XML element that contains other elements and/or attributes. In java we will use Class to create user-defined datatype and in XSD we will use ComplexType to create user-defined data type.

Declaring <order-item> as compound element in XSD

```
<xs:element name="order-item"  type="OrderItemType" />

<xs:complexType name="OrderItemType">
  <xs:sequence>
      <xs:element name="item-code" type="xs:string"/>
      <xs:element name="quantity"  type="xs:int" />
  </xs:sequence>
</xs:complexType>
```

Approach-1

```
<xs:element  name="order-item">
 <xs:complexType>
   <xs:sequence>
       <xs:element name="item-code" type="xs:string" />
       <xs:element  name="quantity"   type="xs:int" />
   </xs:sequence>
 </xs:complexType>
</xs:element>
```

Approach-2

Note: <xs:sequence /> represents child elements should be in same order

Now we understood how to declare Simple Elements and Compound elements in XSD so let's take our PO.xml and write PO.xsd

PO.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<PurchaseOrder>
  <order-items>
    <item>
      <item-code>IC001</item-code>
      <quantity>10</quantity>
    </item>
  </order-items>
  <shipping-address>
    <addrLine1>CDAC Building</addrLine1>
    <addrLine2>S.R.Nagar</addrLine2>
    <city>Hyd</city>
    <state>TG</state>
    <country>India</country>
    <zipcode>500031</zipcode>
  </shipping-address>
</PurchaseOrder>
```

PO.xml

From the above XML identity Compound Elements and Simple Elements and declare them in XSD

PurchaseOrder, order-items, item and shipping-address are compound elements. To declare these compound elements we need Complex Types (For each compound element we need one complex type).

Item-code, quantity, addrLine1, addrLine2, city, state, country and zipcode are Simple Elements

- As XSD is also an XML it will start with Prolog
- XSD contains only one root element that is <xs:schema />
- Inside <xs:schema /> we can declare both Simple elements and compound elements

Below is the PO.xsd which represents structure of PO.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:element name="PurchaseOrder" type="PurchaseOrderType"/>
  <xs:complexType name="PurchaseOrderType">
        <xs:sequence>
                <xs:element name="order-items" type="OrderItemsType"/>
                <xs:element name="shipping-address" type="ShippingAddressType"/>
        </xs:sequence>
  </xs:complexType>
  <xs:complexType name="OrderItemsType">
        <xs:sequence>
                <xs:element name="item" type="ItemType"/>
        </xs:sequence>
  </xs:complexType>
  <xs:complexType name="ItemType">
        <xs:sequence>
                <xs:element name="item-code" type="xs:string"/>
                <xs:element name="quantity" type="xs:int"/>
        </xs:sequence>
  </xs:complexType>
  <xs:complexType name="ShippingAddressType">
        <xs:sequence>
                <xs:element name="addrLine1" type="xs:string"/>
                <xs:element name="addrLine2" type="xs:string"/>
                <xs:element name="city" type="xs:string"/>
                <xs:element name="state" type="xs:string"/>
                <xs:element name="country" type="xs:string"/>
                <xs:element name="zipcode" type="xs:int"/>
        </xs:sequence>
  </xs:complexType>
</xs:schema>
```

PO.xsd

In the approach, first we have taken XML and we have created XSD.

XSD → Represents Structure of the XML

XML → Holds the data

As XSD represents the structure of the XML, first we should create XSD and then we need to create the XML to store the data in the form of elements. Once data is store in XML, we can validate our XML against XSD.

To Validate XML against XSD, we should link XSD with our XML like below

```
<?xml version="1.0" encoding="UTF-8"?>

<rootElement xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xsi:noNamespaceSchemaLocation='fileName.xsd'>

        ..... (child elements goes here)


</rootElement>
```

Xmlns:xsi :- It represents XMLSchema-instance Namespace

Xsi:noNamespaceSchemaLocation :- Represents XSD file name (No namespaces in XSD)

Note : If we have namespaces in XSD then we should use xsi:schemaLocation attribute to link XSD to xml.

Linking PO.xsd to PO.xml

Once XSD is linked to XML, we can validate our xml against XSD using Altova XML Spy tool (using F8 shortcut key).

Element Cardinality

It is possible to constrain the number of instances (cardinality) of an XML element that appear in an XML document. The cardinality is specified using the minOccurs and maxOccurs attributes, and allows an element to be specified as mandatory, optional, or can appear up to a set number of times. The default values for minOccurs and maxOccurs is 1. Therefore, if both the minOccurs and maxOccurs attributes are absent, as in all the previous examples, the element must appear once and once only.

minOccurs can be assigned any non-negative integer value (e.g. 0, 1, 2, 3... etc.), and maxOccurs can be assigned any non-negative integer value or the special string constant "unbounded" meaning there is no maximum so the element can occur an unlimited number of times.

Defining Compositors

Compositors provide rules that determine how and in what order their children can appear within XML document. There are three types of compositors. They are

1. **<xs: sequence>**
2. **<xs: choice>**
3. **<xs: all>**

Compositor	Description
Sequence	The child elements in the XML document MUST appear in the order they are declared in the XSD schema.
Choice	Only one of the child elements described in the XSD schema can appear in the XML document.
All	The child elements described in the XSD schema can appear in the XML document in any order.

<xs: sequence /> element

The <sequence> indicator specifies that the child elements must appear in a specific order:

<xs: all /> element

The <all> indicator specifies that the child elements can appear in any order, and that each child element must occur only once:

<xs: choice /> element

XML Schema <choice> element allows only one of the elements contained in the <choice> declaration to be present within the containing element.

```
<xs:complexType name="ShippingAddressType">

        <xs:choice>
                <xs:element name="office-address" type="OfficeAddressType"/>
                <xs:element name="home-address" type="HomeAddressType"/>
        </xs:choice>

</xs:complexType>

<xs:complexType name="OfficeAddressType">
        <xs:sequence>
                <xs:element name="office-name" type="xs:string"/>
                <xs:element name="office-location" type="xs:string"/>
        </xs:sequence>
</xs:complexType>

<xs:complexType name="HomeAddressType">
        <xs:sequence>
                <xs:element name="addrLine1" type="xs:string"/>
                <xs:element name="addrLine2" type="xs:string"/>
        </xs:sequence>
</xs:complexType>
```

shipping-address.xsd

As per above XSD, ShippingAddress can contain either office-address or home-address but not both

Global elements with ref attribute in XSD

- **If we declare an element inside ComplexType then it is called as local element**
- **If we declare an element inside <xs: schema> directly then that is called as global element.**
- **If Multiple ComplexTypes want to use same name for the element with same data type then we can use Global Elements with ref attribute**

Let's take BookStore.xml to demonstrate this

```
<?xml version="1.0" encoding="UTF-8"?>
<BookStore>
        <Book>
            <name>Spring</name>
            <price>500.00</price>
        </Book>
        <Author>
            <name>Rod Johnson</name>
            <email>rod@gmail.com</email>
        </Author>
</BookStore>
```

BookStore.xml

BookStore.xsd for above BookStore.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:element name="BookStore" type="BookStoreType" />

<xs:complexType name="BookStoreType">
        <xs:sequence>
          <xs:element name="Book" type="BookType"/>
          <xs:element name="Author" type="AuthorType"/>
        </xs:sequence>
</xs:complexType>

<xs:complexType name="BookType">
        <xs:sequence>
          <xs:element name="name" type="xs:string"/>
          <xs:element name="price" type="xs:double"/>
        </xs:sequence>
</xs:complexType>

<xs:complexType name="AuthorType">
        <xs:sequence>
          <xs:element name="name" type="xs:string"/>
          <xs:element name="email" type="xs:string"/>
        </xs:sequence>
</xs:complexType>

</xs:schema>
```

BookStore.xsd(Without global element)

If we see in the above BookStore.xsd, "name" element is declared in 2 complex types (Duplicate element declarations are presented). To avoid duplicate element declarations, we can use Global Element for "name" element like below

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" >

    <xs:element name="name" type="xs:string"/>

    <xs:element name="BookStore" type="BookStoreType"/>

    <xs:complexType name="BookStoreType">
        <xs:sequence>
          <xs:element name="Book" type="BookType"/>
          <xs:element name="Author" type="AuthorType"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="BookType">
        <xs:sequence>
          <xs:element name="id" type="xs:int"/>
          <xs:element ref="name"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="AuthorType">
        <xs:sequence>
          <xs:element ref="name"/>
          <xs:element name="email" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```

BookStore.xsd (with Global element)

<xsd:group> Element

If we want to re-use group of elements in several complex types then we can use <xsd: group> element

Lets' take Bank-Account.xml to demonstrate this

```xml
<?xml version="1.0" encoding="UTF-8"?>
<bank-account>
    <savings-account>
        <acc-id>9130100278</acc-id>
        <holder-name>Ashok</holder-name>
        <branch-name>S.R.Nagar</branch-name>
        <min-bal>3000</min-bal>
    </savings-account>
    <current-account>
        <acc-id>009997979</acc-id>
        <holder-name>Ashok IT</holder-name>
        <branch-name>Hitec city</branch-name>
        <tx-limit>150000</tx-limit>
    </current-account>
</bank-account>
```

bank-account.xml

In the bank-account.xml, both <savings-account /> and <current-account /> having below three elements as common

- <acc-id />
- <holder-name />
- <branch-name />

Instead of writing these 3 elements as global elements, we can create them as a group in xsd like below

```
<xs:group name="AccDetailGroup">
        <xs:sequence>
                <xs:element name="acc-id" type="xs:int"/>
                <xs:element name="holder-name" type="xs:string"/>
                <xs:element name="branch-name" type="xs:string"/>
        </xs:sequence>
  </xs:group>
```

Now we can reference to this “AccDetailGroup” in multiple complextypes to use acc-id , holder-name and branch-name elements.

The modified bank-account.xsd is below (With Group element)

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
        <xs:group name="AccDetailGroup">
                <xs:sequence>
                        <xs:element name="acc-id" type="xs:int"/>
                        <xs:element name="holder-name" type="xs:string"/>
                        <xs:element name="branch-name" type="xs:string"/>
                </xs:sequence>
        </xs:group>
        <xs:element name="bank-account" type="BankAccountType"/>
        <xs:complexType name="BankAccountType">
                <xs:sequence>
                        <xs:element name="savings-account" type="SavingsAccountType"/>
                        <xs:element name="current-account" type="CurrentAccountType"/>
                </xs:sequence>
        </xs:complexType>
        <xs:complexType name="SavingsAccountType">
                <xs:sequence>
                        <xs:group ref="AccDetailGroup"/>
                        <xs:element name="min-bal" type="xs:decimal"/>
                </xs:sequence>
        </xs:complexType>
        <xs:complexType name="CurrentAccountType">
                <xs:sequence>
                        <xs:group ref="AccDetailGroup"/>
                        <xs:element name="tx-limit" type="xs:int"/>
                </xs:sequence>
        </xs:complexType>
</xs:schema>
```

bank-account.xsd

Inheritance in XSD

Complex type extensions are quite similar to the inheritance in Java and other object-oriented languages. New complex types can be derived by extending existing complex types.

```
<xs:complexType name="BaseComplexType">
    <xs:sequence>
        <xs:element name="elementName1" type="dataType"/>
        <xs:element name="elementName2" type="dataType"/>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="DerivedComplexType">
    <xs:complexContent>
        <xs:extension base="BaseComplexType">
            <xs:sequence>
                <xs:element name="elementName3" type="dataType"/>
                <xs:element name="elementName4" type="dataType"/>
            </xs:sequence>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
```

Now let's take an travel-agency.xml to demonstrate this

```
<?xml version="1.0" encoding="UTF-8"?>
<TravelAgency>
    <Passenger>
        <passenger-name>Ashok</passenger-name>
        <passenger-email>ashok@gmail.com</passenger-email>
        <journey-date>2019-08-21</journey-date>
    </Passenger>
    <DomesticFlight>
        <flight-num>DM101</flight-num>
        <pilot-name>Natasha</pilot-name>
        <pilot-contact>9909120899</pilot-contact>
    </DomesticFlight>
    <InternationalFlight>
        <flight-num>IN909</flight-num>
        <pilot-name>James</pilot-name>
        <pilot-contact>08320834</pilot-contact>
        <visa-status>true</visa-status>
    </InternationalFlight>
</TravelAgency>
```

travel-agency.xml

In the above xml, both <DomesticFlight /> and <InternationalFlight /> complextypes are some common elements. So in XSD we can extend one complextype from another to re-use the elements

In below XSD <InternaltionFlight /> complextype is extending elements from <DomesticFlight /> complextype

Attributes in XSD

An attribute provides extra information within an element. Attributes have name and type properties and are defined within an XSD as follows:

```
<xs:attribute name="attributeName" type="datatype" />
```

An attribute is specified within a xs:complexType, the type information for the attribute comes from a xs:simpleType (either defined inline or via a reference to a built in or user defined xs:simpleType definition). The Type information describes the data the attribute can contain in the XML document, i.e. string, integer, date etc. Attributes can also be specified globally and then referenced.

Sample XSD	Sample XML

`<xs:element name="Order">`
`  <xs:complexType>`
`  <xs:attribute name="OrderID"`
`  type="xs:int" />`
`  </xs:complexType>`
`</xs:element>`
	`<Order OrderID="6" />`
	- or no attribute -
	`<Order />`

`<xs:element name="Order">`
`  <xs:complexType>`
`  <xs:attribute name="OrderID"`
`  type="xs:int"`
`  use="optional" />`
`  </xs:complexType>`
`</xs:element>`
	`<Order OrderID="6" />`
	- or no attribute -
	`<Order />`

`<xs:element name="Order">`
`  <xs:complexType>`
`  <xs:attribute name="OrderID"`
`  type="xs:int"`
`  use="required" />`
`  </xs:complexType>`
`</xs:element>`
	`<Order OrderID="6" />`

- Mandatory Attribute: <attribute name= “empNo" type= “int“ use=“required”/>
- Default Attribute: <attribute name= “name” type= “string” default= “guest”/>
- Fixed Attribute: <attribute name= “salary” type= “decimal” fixed= “10000”/>

Note: The default and fixed attributes can be specified within the XSD attribute specification (in the same way as they are for elements).

Element with attributes

An empty complex element means no content rather only attributes.

```
<xs:element name="product">
        <xs:complexType>
                <xs:attribute name="id" type="xs:positiveInteger"/>
        </xs:complexType>
</xs:element>
```

In the example above, we define a complex type with a complex content. The complexContent element signals that we intend to restrict or extend the content model of a complex type, and the restriction of integer declares one attribute but does not introduce any element content.

Example:

<product id=”12345”/>

Element with attributes and text

It contains only text and attributes i.e., it contains only simple content (text and attributes), therefore we add a simpleContent element around the content. When using simpleContent, we must define an extension or restriction within the simpleContent element.

Note: Use the extension/restriction element to expand or to limit the base simple type for the element.

```
<xs:element name="shoesize">
        <xs:complexType>
                <xs:simpleContent>
                        <xs:extension base="xs:integer">
                                <xs:attribute name="country" type="xs:string" />
                        </xs:extension>
                </xs:simpleContent>
        </xs:complexType>
  </xs:element>
```

Example:

<shoesize country="france">35</shoesize>

Element with sub elements and text

An XML element contains both text and sub elements.

```
<xs:element name="letter">
        <xs:complexType mixed="true">
                <xs:sequence>
                        <xs:element name="name" type="xs:string"/>
                        <xs:element name="orderid" type="xs:positiveInteger"/>
                        <xs:element name="shipdate" type="xs:date"/>
                </xs:sequence>
        </xs:complexType>
</xs:element>
```

Example:

<letter> </letter>

Dear Mr.<name>John Smith</name>. Your order <orderid>1032</orderid>.
Will be shipped on <shipdate>2001-07-13</shipdate>.

Sample XSD

XSD	XML
<?xml verson="1.0" encoding="UTF-8"?> <xs:schema> <xs:element name="zipcode" type="xs:int" /> </xs:schema>	<?xml verson="1.0" encoding="UTF-8"?> <zipcode>10</zipcode>

As per the above XSD, given XML is valid or not ?- It is valid only because xsd says zipcode should contain int value and xml having int value only but practically zipcode should not be single digit then how to validate particular element should contain specific No.of digits ?- That's where Restrictions comes into picture in XSD.

Primitive types

The following table summarizes Primitive types in XML Schema Definition:

Primitive type	Description	Example
Decimal	Specifies numeric value including fractional part.	<prize>999.50</prize>
Float		
Double		
Boolean	Specifies true or false value.	
String	The string data type can contain characters, line feeds, carriage returns, and tab characters.	<customer>John Smit</customer>
Date	The date format is always as: YYYY-MM-DD.	<start>2012-01-21</start>
Time	The time format is always as: hh:mm:ss	<time>09:00:00</time>
dateTime	The dateTime format is always as: YYYY-MM-DD**T**hh:mm:ss	<startdate>2012-01-21**T**09:00:00</startdate>
gYearMonth	Defines a part of a date - the year and month (YYYY-MM)	
gYear	Defines a part of a date - the year (YYYY)	
gMonthDay	Defines a part of a date - the month and day (MM-DD)	
gDay	Defines a part of a date - the day (DD)	
gMonth	Defines a part of a date - the month (MM)	
Duration	Defines a time interval	<period>P5Y2M10DT15H</period> The example above indicates a period of five years, two months, 10 days, and 15 hours.
Base64Binary	Base64-encoded binary data	
hexBinary	hexadecimal-encoded binary data	
anyURI		
QName		
NOTATION		

Derived Types: The following table summarized the derived types

Derived type	Description	Example
integer	Derived from decimal type without fractional part.	
Long	Derived from decimal, which is signed 64-bit integer.	
Int	Derived from decimal, which is signed 32-bit integer.	
Short	Derived from decimal, which is signed 16-bit integer.	
Byte	Derived from decimal, which is signed 8-bit integer.	
nonPositiveInteger	An integer containing only non-positive values (..,-2,-1,0)	
negativeInteger	An integer containing only negative values(…,-2,-1)	
nonNegativeInteger	An integer containing only non-negative values (0,1,2,..)	
UnsignedLong	Derived from decimal, which is unsigned 64-bit integer.	
UnsignedInt	Derived from decimal, which is unsigned 32-bit integer.	
UnsignedShort	Derived from decimal, which is unsigned 16-bit integer.	
UnsignedByte	Derived from decimal, which is unsigned 8-bit integer.	
PositiveInteger	An integer containing only positive values(1,2…).	

normalizedString	Derived from string type. Contains only characters without line feeds, carriage returns, and tab characters.	<customer>John Smith</customer>
Token	Derived from string type. Contains only characters without line feeds, carriage returns, tabs, leading and trailing spaces, and multiple spaces.	<customer>John Smith</customer>
Language	A string that contains a valid language id.	
Name	A string that contains a valid XML name.	
NCName	Name without colons but allows characters such as hyphens.	bold_brash is valid. bold:brash is invalid.
ID	Only used with attributes.	
IDREF		
IDREFS		
ENTITY		
ENTITIES		
NMTOKEN	Derived from string, which does not accept Commas, Leading or trailing whitespace will be removed. Used with only attributes.	"x, y" is invalid. "x y" is valid.

Built-in types with restrictions

The built-in types with restrictions are simple types.

Restrictions are used to define acceptable values for XML elements or attributes. Restrictions on XML elements are called **facets**.

Restriction on values

Following example defines an element named "age" with a restriction. The value of the age element cannot be lower than 21 and greater than 60.

```
<xs:element name="age">
        <xs:simpleType>
                <xs:restriction base="xs:integer">
                        <xs:minInclusive value="21"/>
                        <xs:maxInclusive value="60"/>
                </xs:restriction>
        </xs:simpleType>
</xs:element>
```

Restriction on Set of values

To limit the content of an XML element to a set of acceptable values, we would use the enumeration constraint.

```
<xs:element name="car">
        <xs:simpleype>
                <xs:restriction base="xs:string">
                        <xs:enumeration value="Audi"/>
                        <xs:enumeration value="Golf"/>
                        <xs:enumeration value="BMW"/>
                </xs:restriction>
        </xs:simpleType>
</xs:element>
```

Restrictions on Series of Values

The 'pattern' constraint is used to limit the content of xml element to define a series of numbers or letters.

Example #1: The only acceptable value is THREE of the LOWERCASE or UPPERCASE letters from a to z or A to Z.

```
<xs:element name="initials">
        <xs:simpleType>
                <xs:restriction base="xs:string">
                        <xs:pattern value="[a-zA-Z][a-zA-Z][a-zA-Z]"/>
                </xs:restriction>
        </xs:simpleType>
</xs:element>
```

Example #2: The acceptable value is zero or more occurrences of lowercase letters from a to z.

```
<xs:element name="letter">
        <xs:simpleType>
                <xs:restriction base="xs:string">
                        <xs:pattern value="([a-z])*"/>
                </xs:restriction>
        </xs:simpleType>
</xs:element>
```

Example #3: The only acceptable value is male OR female.

```
<xs:element name="gender">
        <xs:simpleType>
                <xs:restriction base="xs:string">
                        <xs:pattern value="male|female"/>
                </xs:restriction>
        </xs:simpleType>
</xs:element>
```

Restrictions on Length

To limit the length of a value in an element, we would use the length, maxLength, and minLength constraints.

Example #1: This example defines an element called "password" with a restriction. The value must be exactly eight characters:

```
<xs:element name="password">
        <xs:simpleType>
                <xs:restriction base="xs:string">
                        <xs:length value="8"/>
                </xs:restriction>
        </xs:simpleType>
</xs:element>
```

Example #2: This example defines another element called "password" with a restriction. The value must be minimum five characters and maximum eight characters:

```
<xs:element name="password">
        <xs:simpleType>
                <xs:restriction base="xs:string">
                        <xs:minLength value="5"/>
                        <xs:maxLength value="8"/>
                </xs:restriction>
        </xs:simpleType>
</xs:element>
```

The following table summarizes attributes related to restrictions

CONSTRAINT	DESCRIPTION
Enumeration	Defines a list of acceptable values
Length	Specifies the exact number of characters or list items allowed. Must be equal to or greater than zero
maxLength	Specifies the maximum number of characters or list items allowed. Must be equal to or greater than zero.
minLength	Specifies the minimum number of characters or list items allowed. Must be equal to or greater than zero
maxExclusive	Specifies the upper bounds for numeric values (the value must be less than this value)
maxInclusive	Specifies the upper bounds for numeric values (the value must be less than or equal to this value)
minExclusive	Specifies the lower bounds for numeric values (the value must be greater than this value)
minInclusive	Specifies the lower bounds for numeric values (the value must be greater than or equal to this value)
Pattern	Defines the exact sequence of characters that are acceptable
totalDigits	Specifies the exact number of digits allowed. Must be greater than zero
fractionDigits	Specifies the maximum number of decimal places allowed. Must be equal to or greater than zero

<any>

The <any> element enables us to extend the XML document with elements not specified by our schema.

#family.xsd

```
<xs:schema>
        <xs:element name="person">
                <xs:complexType>
                        <xs:sequence>
                                <xs:element name="firstname" type="xs:string"/>
                                <xs:element name="lastname" type="xs:string"/>
                                <xs:any minOccurs="0" maxOccurs="1"/>
                        </xs:sequence>
                </xs:complexType>
        </xs:element>
</xs:schema>
```

Now we want to extend the "person" element with a "children" element. In this case we can do so, even if the author of the schema above never declared any "children" element.

#children.xsd

```
<xs:schema>
  <xs:element name="children">
        <xs:complexType>
              <xs:sequence>
                    <xs:element name="childname" type="xs:string" maxOccurs="unbounded"/>
              </xs:sequence>
             </xs:complexType>
       </xs:element>
</xs:schema>
```

The XML file below (called "Myfamily.xml"), uses components from two different schemas; "family.xsd" and "children.xsd".

#Myfamily.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persons>
        <person>
                <firstname>Ashok</firstname>
                <lastname>Kumar</lastname>
                <children>
                        <childname>Sachin</childname>
                        <childname>Sehwag</childname>
                </children>
        </person>
</persons>
```

So far in the XSD we have ignored namespaces as they will create confusion while writing and using basic XSDs. Just to make you feel comfortable with XSD i have ignored namespaces in XSD but we can't ignore namespaces completely. Namespaces plays vital role when we are working XSD and namespaces are one of the reason to use XSD inplace of DTD. Now let’s see what is namespace and what is the purpose of Namespace.

XSD Namespaces

When we are working on a project, several developers will create several classes and every developer will have a freedom to choose their interested name for a class. In this scenario is there any possibility that multiple developers will create classes using same name? – Yes, possibility is there so it causes naming collision. Then how they can differentiate this class belongs to which developer? – That's where java provided Packages concept.

Packages are used to resolve naming collision that occurs across multiple classes which are created by multiple developers.

To resolve the naming collision every developer will bind his/her class to a package like below

```
package   com.ibm.ctod.reports.dao ;

public class SummaryReportDao {

        //methods goes here

}
```

SummaryReportDao.java

With this SummaryReportDao.java class is binded to **com.ibm.ctod.reports.dao** package. So, wherever we want to use **SummaryReportDao.java** class we should fully qualified class name like below

```
package com.ibm.ctod.reports.service {

        public  void generateReport ( ) {

                com.ibm.ctod.reports.dao.SummaryReportDao  srdao =
                        new com.ibm.ctod.reports.dao.SummaryReportDao ( ) ;

                  srdao.methodName ( ) ;
        }
}
```

Note: Instead of writing **com.ibm.ctod.reports.dao** multiple times in class, we can import this package using import statement.

Like java, when multiple people are working on XSDs there is a possibility that they will use same names for elements and ComplexTypes then it causes naming collision. To resolve this naming collision XSD provided Namespaces.

XSD Namespaces has two faces

- Declaring the namespace in the XSD document using Target namespace declaration
- Using the elements that are declared under a namespace in xml document.

XSD Target Namespace

Target namespace declaration is similar to a package declaration in java. You will bind your classes to a package so, that while using them you will refer with fully qualified name. Similarly, when you create a ComplexType or an Element you will bind them to a namespace, so that while referring them you need to use qName.

In order to declare a package, we use package keyword followed by name of the package. To declare a namespace in XSD we need to use targetNamespace attribute at the Schema level followed by targetNamespace label as shown below.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
3  xmlns:airtel="http://www.airtel.in/payments"
4  targetNamespace="http://www.airtel.in/payments"
5  elementFormDefault="qualified">
6
7     <xs:complexType name="PaytmType">
8        <xs:sequence>
9           <xs:element name="consumer-name" type="xs:string"/>
10          <xs:element name="charges" type="xs:double"/>
11       </xs:sequence>
12    </xs:complexType>
13
14    <xs:element name="paytm" type="airtel:PaytmType"/>
15 </xs:schema>
```

Paytm.xsd

The PaytmType is by default is binded to the namespace **"http://www.airtel.in/payments"**

So, while creating an element "paytm" of type PaytmType we should use the qName of the PaytmType rather simple name. (qName means namespace:element/type name).

But the namespace labels could be any of the characters in length, so if we prefix the entire namespace label to element/type names it would become tough to read. So, to avoid this problem XSD has introduced a concept called short name. Instead of referring to namespace labels you can define a short name for that namespace label using xmlns declaration at the <schema> level and you can use the short name as prefix instead of the complete namespace label. In above Paytm.xsd we have written "airtel" as alias name for targetNamespace like below

```
xmlns:airtel="http://www.airtel.in/payments"
targetNamespace="http://www.airtel.in/payments"
```

In java we can have only one package declaration at a class level. In the same way we can have only one targetNamespace declaration at an XSD document.

Importance of xs:include and xs:import

Till now we have assumed that we only have a single schema file containing all your element definitions, but the XSD standard allows you to structure your XSD schemas by breaking them into multiple files. These child schemas can then be included into a parent schema.

Breaking schemas into multiple files can have several advantages. You can create re-usable definitions that can be used across several projects. They make definitions easier to read and version as they break down the schema into smaller units that are simpler to manage.

When schemas breaked into multiple files how one schema can refer element/complexType from another schema? – That's where <xs:inclue /> and <xs:import /> comes into picture.

<xs:include /> : If you want to use XML schema components (elements/ ComplexTypes) from other XML schemas with same targetNameSPaces, we should go for xs:include element

Syntax: <xs:include schemaLocation="SchamUri"/>
Ex : <xs:include schemaLocation ="Invoice.xsd" />

Let's demonstrate this using below Example

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
3  targetNamespace="http://www.flipkart.com/sales"
4  xmlns:fps="http://www.flipkart.com/sales" elementFormDefault="qualified">
5    <xs:complexType name="InvoiceType">
6      <xs:sequence>
7        <xs:element name="invoice-id" type="xs:int"/>
8        <xs:element name="bill-amount" type="xs:double"/>
9        <xs:element name="generated-dt" type="xs:date"/>
10     </xs:sequence>
11   </xs:complexType>
12   <xs:element name="invoice-info" type="fps:InvoiceType"/>
13 </xs:schema>
```

Invoice.xsd

Inovice.xsd is included in Customer.xsd using <xs:include /> element

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
3      targetNamespace="http://www.flipkart.com/sales"
4      xmlns:fps="http://www.flipkart.com/sales"
5      elementFormDefault="qualified">
6    <xs:include schemaLocation="C:\Ashok\Invoice.xsd"/>
7    <xs:complexType name="CustomerType">
8      <xs:sequence>
9        <xs:element name="fist-name" type="xs:string"/>
10       <xs:element name="last-name" type="xs:string"/>
11       <xs:element name="email" type="xs:string"/>
12       <xs:element ref="fps:invoice-info"/>
13     </xs:sequence>
14   </xs:complexType>
15   <xs:element name="customer-info" type="fps:CustomerType"/>
16 </xs:schema>
```

Customer.xsd

We can write the XML for Customer.xsd like below

<xs:import /> :- If we want to use XML schema components from other XML schamas with different targetNameSpaces, we should go for xs:import element.

Syntax : <xs:import namespace="targetnamespace" schemalocation="xsd uri"/>
Ex : <xs:import namespace=http://amazon.com/books/webservices schemaLocation="amz.xsd"/>

Let's demonstrate <xs: import /> this with below example

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
3  xmlns:sun="http://www.sun.com/authors"
4  targetNamespace="http://www.sun.com/authors"
5  elementFormDefault="qualified">
6      <xs:complexType name="AuthorType">
7          <xs:sequence>
8              <xs:element name="author-name" type="xs:string"/>
9              <xs:element name="author-email" type="xs:string"/>
10         </xs:sequence>
11     </xs:complexType>
12     <xs:element name="author-info" type="sun:AuthorType"/>
13 </xs:schema>
```

Author.xsd

Author.xsd is imported into Book.xsd using <xs: import /> element

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
3  xmlns:ebs="http://www.ebay.in/sales"
4  xmlns:sun="http://www.sun.com/authors"
5  targetNamespace="http://www.ebay.in/sales"
6  elementFormDefault="qualified">
7
8      <xs:import namespace="http://www.sun.com/authors"
9          schemaLocation="C:\Ashok\Author.xsd"/>
10
11     <xs:complexType name="BookType">
12         <xs:sequence>
13             <xs:element name="book-title" type="xs:string"/>
14             <xs:element name="isbn" type="xs:string" nillable="true"/>
15             <xs:element name="price" type="xs:double"/>
16             <xs:element ref="sun:author-info"/>
17         </xs:sequence>
18     </xs:complexType>
19     <xs:element name="book-info" type="ebs:BookType"/>
20 </xs:schema>
```

Book.xsd

Below is the xml that represents Book.xsd

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <ebs:book-info xmlns:ebs="http://www.ebay.in/sales"
3  xmlns:sun="http://www.sun.com/authors"
4  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5  xsi:schemaLocation="http://www.ebay.in/sales  file:///C:/Ashok/Book.xsd">
6      <ebs:book-title>Webservices</ebs:book-title>
7      <ebs:isbn>ISBN001</ebs:isbn>
8      <ebs:price>1500.00</ebs:price>
9      <sun:author-info>
10         <sun:author-name>Ashok</sun:author-name>
11         <sun:author-email>ashok.javatraining@gmail.com</sun:author-email>
12     </sun:author-info>
13 </ebs:book-info>
```

Book-Author.xml

JAX-P

Introduction

JAX-P stands for Java API for XML Processing. As we discussed earlier, XML is also a document which holds data in an XML format. Initially in Java there is no API that allows reading the contents of an XML document in an XML format. Java programmers has to live with Java IO programming to read them in a text nature or need to build their own logic to read them in XML format.

By looking at this lot of Java software vendors in the market has started building their own libraries/API's that allows processing an XML documents in XML nature. Few of them are DOM4J, JDOM.

After many software vendors started developing their own libraries, sun has finally released JAX-P API which allows us to work with XML documents. As indicated JAX-P is an API which means not complete in nature, so we need implementations of JAX-P API and there are many implementations of JAX-P API available for e.g.. Crimson, Xerces2, Oracle V2 Parser etc.

Xerces2 is the default parser that would be shipped as part of JDK1.5+. This indicates if you are working on JDK1.5+ you don't need any separate Jar's to work with JAX-P API.

XML Processing Methodologies

JAXP provides wrappers around two different mechanisms for processing XML data.The first is the Simple API for XML or SAX, and is covered in this chapter. The second, the Document Object Model (DOM), is covered in the next. In the SAX model, XML documents are provided to the application as a series of events, with each event representing one transition in the XML document. For example, the beginning of a new element counts as an event, as does the appearance of text inside that element. A SAX parser reads through the XML document one time, reporting each event to the application exactly once in the order it appears in the document. Event-based parsing has strengths and weaknesses.Very large documents can be processed with events; there is no need to read the entire document into memory at once. However, working with sections of an XML document (a record made up of many elements, for example) can become complicated because the application developer has to track all the events for a given section. SAX is a widely used standard, but is not controlled by any industry group. Rather, it is a de facto standard that was originally developed by a single developer (David Megginson) and by others in the XML community and is now supported by an open source project (http://www.saxproject.org). The SAX wrapper provided by JAXP allows for plugging in different SAX parsers without concern for the underlying implementation.This feature is somewhat moot because there aren't that many SAX parsers in widespread use today. However, it does provide for safeguards against changes in future versions. SAX and DOM are the universal methodologies of processing (reading) XML documents irrespective of a particular technology. Few programming languages/API's supports only SAX way of reading documents, few other supports only DOM and quite a few number support both the ways of reading.

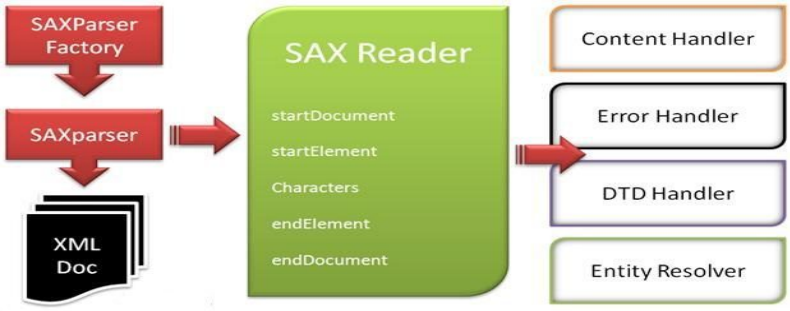
Simple API for XML (SAX)

The Simple API for XML (SAX) is available with the JAXP; SAX is one of two common ways to write software that accesses XML data. SAX is an event-driven methodology for XML processing and consists of many callbacks. Using SAX with JAXP allows developers to traverse through XML data sequentially, one element at a time, using a delegation even model. Each time elements of the XML structure are encountered, an event is triggered. Developers write event handlers to define customer processing for events they deem important. Each element is parsed down to its leaf node before moving on to the next sibling of that element in the XML document, therefore at no point is there any clear relation of what level of the tree we are at.

The SAX Event Model

Most XML parsers fall into one of two main categories: tree-based or event based. Each kind of parser represents XML information slightly differently. A tree-based parser converts an XML document into a tree of objects (You will learn more about tree-based parsers in the next chapter). An event-based parser presents a document as a series of events, each representing a transition in the document. Taken together, these events provide your program with a complete picture of the document. Overview of Event Processing Imagine again that we have printed out an XML document. How would you begin to read the document? You would probably begin at the top of the page and continue line by line, left to right. Since you are familiar with XML, you know that the tags have specific meanings. You would likely take notice of where one element ends and another begins; and with each new element you would gain a better understanding of the complete document. Event-based parsers work in much the same way. An event-based parser scans through a document from top to bottom. As it scans, the parser takes notice of interesting points in the document.

For example, it would notice where one element ends and another begins. The parser then alerts your program, giving it the opportunity to respond to the transition. In parser terminology, the alert is called an event and your program's response is a callback. A complete document parse may consist of hundreds or even thousands of events in series. Each event provides further information about the document. Handling hundreds or thousands of events sounds overwhelming, and there's no doubt that it can get complicated. Event-based parsing suits some situations better than others. The simpler the DTD associated with a document is, the easier it will be to use event-based processing. Oftentimes these documents contain repeating groups of elements, where each group is to be processed the same way. Poor candidate documents are much the opposite. They are loosely-structured or contain deep and complex hierarchies. You may also want to avoid documents where elements are reused throughout multiple levels of a hierarchy. For these types of documents, consider the tree-based parsers discussed elsewhere in this book. If you are interested in using an event-based parser, then you will need to learn the SAX API.



SAX Parser

Any event based processing model contains three actors as described below.

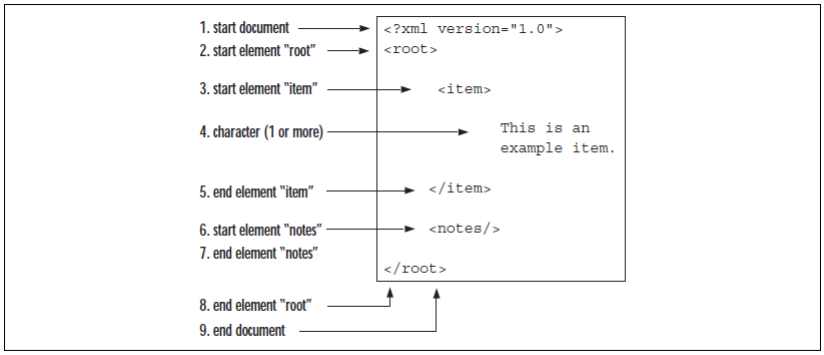
Source: - is the originator of event, who can raise several types of events. Events could be of multiple types if we consider AWT, we can consider source as a button, which is capable of raising several types of events like button click, mouse move, key pressed etc.

Listener: - Listener is the person who will listens for an event from the source. Listener listens for a specific type of event from the source and triggers an even handling method on the Event handler to process it.

Event Handler: - Once a listener captures an event, to process it, it calls a method on the event handler class. As there are different types of events, to handle them the event handler contains several methods each target to handle and process a specific type of event.

In addition, SAX is a sequential processing model, which process XML elements sequentially from top to the bottom. While processing the document, it places a pointer to the document and increments sequentially from the top. Based on the type of element it is pointing to, it raises a respective event, which will notify the listener to handle and process it.

So here source is the XML document which can raise several types of events based on the type of element it is pointing to (e.g... START_DOCUMENT, START_ELEMENT etc...). Listener is the parser who will reads the XML document and triggers a method call on the Handler. Event handler contains several methods to handle various types of events as shown below.



SAX is very fast in processing XML documents when compared with DOM and consumes less amount of memory, as at any given point of time it loads only one element into the memory and process. Here one key thing to note is using SAX we can only read the XML document, we cannot modify/create a document.

Using JAX-P API we can process XML documents in SAX model, here also source is the XML document, who is capable of raising multiple types of events likes startDocument, startElement, endDocument etc. In order to read the elements of XML and process them by raising events, we need a parser and here in SAX it is SAXParser.

When to Use SAX

- You want to process the XML document in a sequential manner from top to bottom.
- SAX requires much less memory than DOM because SAX does not create an in-memory tree of the XML data, as a DOM does. So if you want to process a very large XML document whose DOM tree would consume too much memory, you can shoose SAX over DOM.
- If the XML document is not deeply nested
- SAX is fast and efficient and it is useful for state-independent filtering. SAX parser calls a method whenever an element tag is encountered and calls a different method when text or character is found.

Note: The disadvantage of SAX is that it does not provide random access to an XML document.

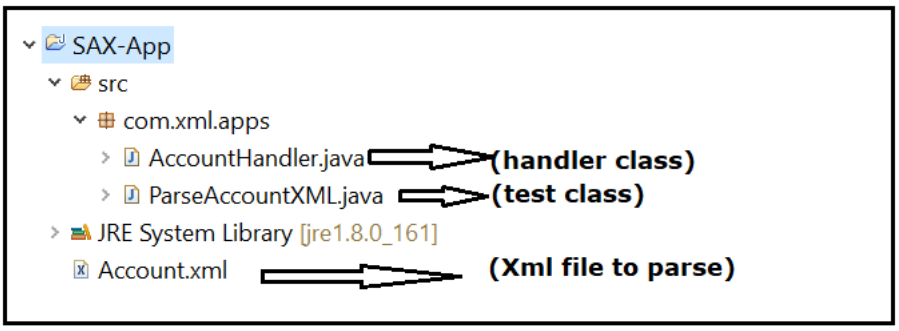
Let us see an example program using SAX parser in Java.

This the xml file that we are going to read using SAX parser. The xml file contains a list of accounts

```
<?xml version="1.0" encoding="UTF-8"?>
<Bank>
    <Account type="saving">
        <Id>1001</Id>
        <Name>John</Name>
        <Amt>10000</Amt>
    </Account>
    <Account type="current">
        <Id>1002</Id>
        <Name>Smith corporation</Name>
        <Amt>1000000</Amt>
    </Account>
</Bank>
```

Account.xml

Create a standalone project to read the above xml



The next step is to create our own handler class to parse the XML document. DefaultHandler class provides default implementation of ContentHandler interface, so we can extend this class to create our own handler.

----------**AccountHandler.java**----------

```
package com.xml.apps;

import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;

public class AccountHandler extends DefaultHandler {

	boolean bank = false;
	boolean account = false;
	boolean id = false;
	boolean name = false;
	boolean amt = false;

	@Override
	public void startDocument() throws SAXException {
		System.out.println("Document started....");
	}

	@Override
	public void startElement(String uri, String localName, String qName,
			Attributes attributes) throws SAXException {
		if (qName.equals("Bank")) {
			bank = true;
		} else if (qName.equals("Account")) {
		account = true;
		  System.out.println("Account Type:" + attributes.getValue("type"));
		} else if (qName.equals("Id")) {
			id = true;
		} else if (qName.equals("Name")) {
			name = true;
		} else if (qName.equals("Amt")) {
			amt = true;
		}
	}

	@Override
	public void characters(char[] ch, int start, int end) throws SAXException {
		if (id) {
			System.out.println("Acct id: " + new String(ch, start, end));
			id = false;
		}
		if (name) {
			System.out.println("Holder Name: " + new String(ch, start, end));
```

```
                name = false;
            }

            if (amt) {
                System.out.println("Amount: " + new String(ch, start, end));
                amt = false;
            }
        }

        @Override
        public void endElement(String uri, String localName, String qName)
                throws SAXException {
        }

        @Override
        public void endDocument() throws SAXException {
            System.out.println("Document ended....");
        }
}
```

----------0----------

In the above class we are overriding five methods which would be triggered based on the type of elements the parser is pointing to on the source XML document.

- startDocument – would be triggered at the start of the XML document
- startElement – Whenever the parser encounters the starting element, it raises this method call by passing the entire element information to the method.
- characters – This method would be invoked by the parser, whenever it encounters data portion between the XML elements. To this method it passes the entire XML as character array along with two other integers one indicating the position in the array from which the current data portion begins and the second integer representing the number of characters the data span to.
- endElement – would be triggered by the parser, when it encounters a closing element or ending element.
- endDocument – would be triggered by the parser once it reaches end of the document.

Below is the Java program that uses SAXParserHandler to parse the XML to list of Employee objects.

ParseAccountXML.java

```java
package com.xml.apps;

import java.io.File;
import java.io.FileInputStream;

import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;

public class ParseAccountXML {

    public static void main(String[] args) throws Exception {

        // Creating SAXParserFactory using factory design pattern
        SAXParserFactory factory = SAXParserFactory.newInstance();

        // Creating SAXParser from factory
        SAXParser parser = factory.newSAXParser();

        // Locating File path
        File f = new File("Account.xml");
        FileInputStream fis = new FileInputStream(f);

        // Parsing xml using AccountHanlder
        parser.parse(fis, new AccountHandler());
    }
}
```

In the last chapter we discussed event-based parsing. Event-based parsing has some limitations. For instance, one cannot manipulate entire sections of a document at once. Also, multiple passes over the document data calls for multiple runs through the parser.Often, it would be more efficient if an entire document could be read into the memory and manipulated as an in-memory representation. This can be achieved using the Document Object Model (DOM), which is by far the most widely used among the various standards proposed for in memory

representation of XML documents. A standard specified by the World Wide Web Consortium (W3C), the DOM specifies an abstract mapping of XML elements into runtime objects. Binding this specification to various languages, including Java, has been implemented by a variety of vendors. In the DOM, every XML document is represented as a hierarchy of objects. This hierarchy forms a tree structure that mimics the structure of the XML document. Once you are familiar with XML and DOM, translating between the two becomes a simple matter.

One thing the current DOM specification omits is an API for the parsing of an XML document into a DOM. This is left to the individual vendor when writing a DOM parser. The latest DOM specification (DOM 3) seeks to address

this issue too, but until it gains in popularity, application developers have to take into account the prospects of modifying their code when moving to a different parser. JAXP solves this problem by presenting a standardized interface for parsing

XML data; the result of this parsing is an object that conforms to the standard W3C DOM Document interface.

This chapter presents a discussion on the reading, writing, and simple manipulations of a DOM representation. More complex manipulations, such as those pertaining to stylesheets, will be discussed in the next chapter.

DOM parser

DOM (Document Object Model) defines a standard for accessing and manipulating documents. DOM builds an in-memory tree representation of the XML document, where each node contains one of the components from an XML structure. The two most common types of nodes in XML document are element nodes and text nodes. Java DOM parser API allows us to create nodes, remove nodes, change their contents, and traverse the node hierarchy.

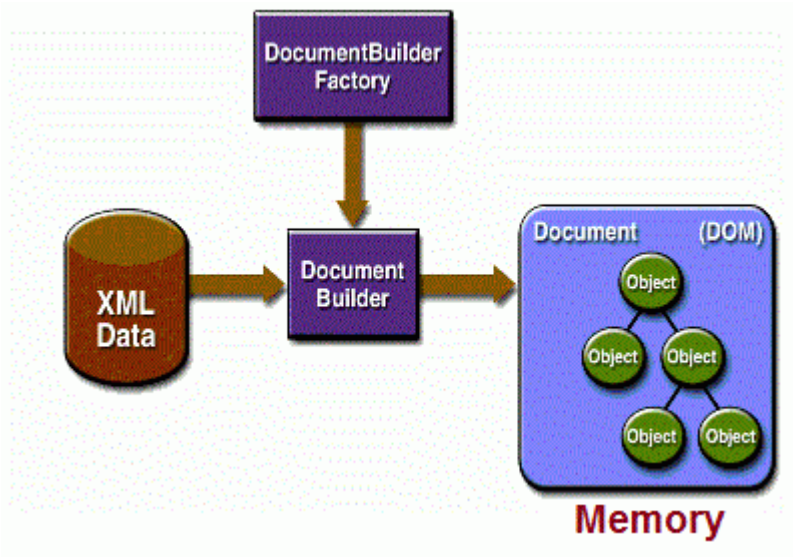
Points to note about DOM

- DOM was designed to be language-neutral.
- DOM does not take advantage of Java's object-oriented features.
- DOM provides a lot of flexibility for handling fully-fledged documents and complex applications. If your programs handle simple data structures, then you can use JDOM or dom4j.
- Since DOM creates in-memory tree, for processing very large XML documents, you should choose SAX or StAX beacuse DOM would consume too much memory.

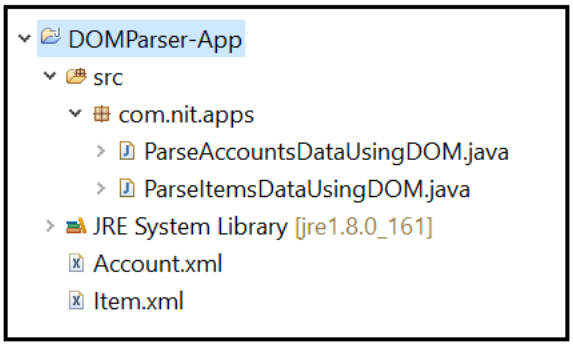
DOM Parser API

The javax.xml.parsers.DocumentBuilder class defines API to obtain DOM Document instances from an XML document. Using this class, an application program can obtain a Document from XML.
An instance of this class can be obtained from the DocumentBuilderFactory.newDocumentBuilder()method. Once an instance of this class is obtained, XML can be parsed from a variety of input sources. These input sources are InputStreams, Files, URLs, and SAX InputSources.

Below is the architecture of a typical DOM parser application.



Parse XML document using DOM parser in Java




```xml
<?xml version="1.0" encoding="UTF-8"?>
<item>
     <itemCode>IC001</itemCode>
     <quantity>10</quantity>
</item>
```

Item.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Bank>
   <Account type="saving">
      <Id>1001</Id>
      <Name>John</Name>
      <Amt>10000</Amt>
   </Account>
   <Account type="current">
      <Id>1002</Id>
      <Name>Smith corporation</Name>
      <Amt>1000000</Amt>
   </Account>
</Bank>
```

Account.xml

----------ParsingAccountsDataUsingDOM.java----------

```
public class ParseAccountsDataUsingDOM {
	public static void main(String[] args) throws ParserConfigurationException,
			SAXException, IOException {
		// Get Document Builder
		DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
		DocumentBuilder builder = factory.newDocumentBuilder();

		// Build Document
		Document document = builder.parse(new File("Account.xml"));

		// Normalize the XML Structure; It's just too important !!
		document.getDocumentElement().normalize();

		// Here comes the root node
		Element root = document.getDocumentElement();
		System.out.println(root.getNodeName());

		// Get all employees
		NodeList nList = document.getElementsByTagName("Account");
		System.out.println("============================");
		visitChildNodes(nList);
	}

	// This method is called recursively
	private static void visitChildNodes(NodeList nList) {
		for (int temp = 0; temp < nList.getLength(); temp++) {
			Node node = nList.item(temp);
			if (node.getNodeType() == Node.ELEMENT_NODE) {
				System.out.println("Node Name = " + node.getNodeName()
						+ "; Value = " + node.getTextContent());
				// Check all attributes
				if (node.hasAttributes()) {
					// get attributes names and values
					NamedNodeMap nodeMap = node.getAttributes();
```

```
                                for (int i = 0; i < nodeMap.getLength(); i++) {
                                        Node tempNode = nodeMap.item(i);
            System.out.println("Attr name:"+ tempNode.getNodeName() + "; Value = "
                                        + tempNode.getNodeValue());
                        }
                        if (node.hasChildNodes()) {
                                // We got more childs; Let's visit them as well
                                visitChildNodes(node.getChildNodes());
                        }
                        }
                }
            }
        }
}
```

Create XML document using DOM parser in Java

Here is the input xml file we will be creating using DOM parser.

```
<?xml version="1.0" encoding="UTF-8"?>
<Employees>
   <Employee id="1">
      <age>28</age>
      <firstname>Ashok</firstname>
      <lastname>Kumar</lastname>
      <role>Developer</role>
   </Employee>
</Employees>
```

Employees.xml

==========DomparserCreateXml.java==========

```
import java.io.File;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;

import org.w3c.dom.Attr;
import org.w3c.dom.Comment;
import org.w3c.dom.Document;
import org.w3c.dom.Element;

public class DomParserCreateXml {
   public static void main(String[] args) {
      try {
          DocumentBuilderFactory dbFactory = DocumentBuilderFactory.newInstance();
          DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
          Document doc = dBuilder.newDocument();
          // root element
          Element rootElement = doc.createElement("Employees");
          doc.appendChild(rootElement);

          // Employee element
          Element employee = doc.createElement("Employee");
          rootElement.appendChild(employee);

          // setting attribute to element
          Attr attr = doc.createAttribute("id");
          attr.setValue("1");
          employee.setAttributeNode(attr);
```

```
            // age element
            Element age = doc.createElement("age");
            age.appendChild(doc.createTextNode("28"));
            employee.appendChild(age);

            // firstname element
            Element firstname = doc.createElement("firstname");
            firstname.appendChild(doc.createTextNode("Ashok"));
            employee.appendChild(firstname);

            // lastname element
            Element lastname = doc.createElement("lastname");
            lastname.appendChild(doc.createTextNode("Kumar"));
            employee.appendChild(lastname);

            // role element
            Element role = doc.createElement("role");
            role.appendChild(doc.createTextNode("Developer"));
            employee.appendChild(role);

            Comment comment = doc.createComment("This is a comment");
            rootElement.appendChild(comment);

            // write the content into xml file
            TransformerFactory transformerFactory = TransformerFactory.newInstance();
            Transformer transformer = transformerFactory.newTransformer();
            /* DOMSource acts as a holder for a transformation Source tree in the
form of a
             * Document Object Model (DOM) tree.
             */
            DOMSource source = new DOMSource(doc);
            StreamResult result = new StreamResult(new File("D:\\employee.xml"));
            transformer.transform(source, result);
            // Output to console for testing purpose
            StreamResult consoleResult = new StreamResult(System.out);
            transformer.transform(source, consoleResult);
        } catch (Exception e) {
                e.printStackTrace();
        }
    }
}
```

StAX parser

StaX (Streaming API for XML) is a JAVA based API to parse XML documents similar to a SAX parser. There are two main differences between SAX and StaX parsers. They are

SAX is a push API: - It means that the SAX parser will send (push) notifications to the client program whenever it has processed an element, attribute or character.

On the other hand StAX is a pull API: - It means the client application need to ask StAX parser to get information from XML whenever it needs.

StAX is an API for reading and writing XML Documents. SAX can be used only for reading XML.

Note that similar to SAX, StAX can only process the XML document in a linear fashion from top to bottom and you cannot have random access to an XML document.

StAX API

StaX contains two distinct APIs to work with XML documents. They are

- Cursor API
- Event Iterator API

Applications can use any of these two API for parsing XML documents.

Cursor API

As the name suggests, the StAX cursor API represents a cursor with which you can parse an XML document from beginning to end. This cursor can point to one thing at a time, and always moves forward, never backward, usually one info set element at a time.

The two main cursor interfaces are

- XMLStreamReader
- XMLStreamWriter

The XMLStreamReader interface has methods to read an XML document
The XMLStreamWriter interface provides methods for creating an XML document.

The XMLStreamReader interface has the following important methods

- **int next()** – used to retrieve next parsing event.
- **boolean hasNext()** – Returns true if there are more parsing events and false if there are no more events.
- **String getText()** – used to get text current value of an element
- **String getLocalName()** – used to get (local)name of an element

The XMLStreamWriter interface has the following important methods.

- **writeStartElement (String localName)** – Add start element of given name.
- **writeEndElement (String localName)** – Add an end element of given name.
- **writeAttribute (String localName, String value)** – Writes an attribute to the output stream without a prefix.
- **writeCharacters (String text)** – Write text to the output.

Read XML using XMLStreamReader

```
------------------------StaxParserExample.java----------------------------
import java.io.FileInputStream;
import java.io.FileNotFoundException;

import javax.xml.stream.XMLInputFactory;
import javax.xml.stream.XMLStreamConstants;
import javax.xml.stream.XMLStreamException;
import javax.xml.stream.XMLStreamReader;

public class StAXParserExample {
   public static void main(String[] args) {
      boolean bFirstName = false;
      boolean bLastName = false;
      boolean bRole = false;
      boolean bAge = false;
      try {
```

```
            XMLInputFactory factory = XMLInputFactory.newInstance();
            XMLStreamReader streamReader = factory.createXMLStreamReader(new
FileInputStream(
"C:/Users/Ashok/Desktop/employees.xml"));
            while (streamReader.hasNext()) {
                  int eventType = streamReader.next();
                  switch (eventType) {
                     case XMLStreamConstants.START_ELEMENT:
                          String qName = streamReader.getLocalName();
                          if (qName.equalsIgnoreCase("Employee")) {
                              System.out.println("Start Element : " + qName);
                              String id = streamReader.getAttributeValue(0);
                              System.out.println("Id : " + id);
                          } else if (qName.equalsIgnoreCase("age")) {
                              bAge = true;
                          } else if (qName.equalsIgnoreCase("firstname")) {
                              bFirstName = true;
                          } else if (qName.equalsIgnoreCase("lastname")) {
                              bLastName = true;
                          } else if (qName.equalsIgnoreCase("role")) {
                              bRole = true;
                          }
                          break;
                     case XMLStreamConstants.CHARACTERS:
                          String characters = streamReader.getText();
                          if (bAge) {
                              System.out.println("Age: " + characters);
                              bAge = false;
                          }
                          if (bFirstName) {
                              System.out.println("First Name: " + characters);
                              bFirstName = false;
                          }
                          if (bLastName) {
                              System.out.println("Last Name: " + characters);
                              bLastName = false;
                          }
                          if (bRole) {
                              System.out.println("Role: " + characters);
                              bRole = false;
                          }
                          break;
                     case XMLStreamConstants.END_ELEMENT:
                          String endName = streamReader.getName().getLocalPart();
                          if (endName.equalsIgnoreCase("Employee")) {
                             System.out.println("End Element :" + endName);
                             System.out.println();
                          }
                          break;
                  }
            }
         } catch (FileNotFoundException | XMLStreamException e) {
                  e.printStackTrace();
      }
   }
```

---.---

Validating XML against XSD

An XML document is considered 'well-formed' if it follows the normal rules of XML. i.e. all tags are closed properly etc. On the other hand, an XML is considered valid if it follows the rules specified in the DTD or XSD.

Below is the Account.xsd which represents Structure of the XML

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

      <xs:element name="Bank" type="BankType" />

      <xs:complexType name="BankType">
        <xs:sequence>
            <xs:element name="Account" type="AccountType" maxOccurs="unbounded" />
        </xs:sequence>
      </xs:complexType>

      <xs:complexType name="AccountType">
        <xs:sequence>
            <xs:element name="id" type="xs:int" />
            <xs:element name="Name" type="xs:string" />
            <xs:element name="Amt" type="xs:double" />
        </xs:sequence>
         <xs:attribute name="type" type="xs:string" />
      </xs:complexType>
</xs:schema>
```

Account.xsd

Below is the Account.xml which holds the data as per Account.xsd .

```
<?xml version="1.0" encoding="UTF-8"?>
<Bank>
      <Account type="saving">
            <Id>1001</Id>
            <Name>1234</Name>
            <Amt>10000</Amt>
      </Account>
      <Account type="current">
            <Id>1002</Id>
            <Name>Smith corporation</Name>
            <Amt>1000000</Amt>
      </Account>
</Bank>
```

Account.xml

Manually we can validate XML against XSD using Altova XML Spy tool but can we validate XML against XSD programmatically? – Yes, we can validate XML against XSD programmatically using Schema and Validator like below

Validating XML against XSD using DOM Parser

```
private void validateXSDDom() throws IOException, ParserConfigurationException {
      try {
          DocumentBuilder builder = DocumentBuilderFactory.newInstance().newDocumentBuilder();
          Document doc = builder.parse("Account.xml");
          SchemaFactory factory = SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
          Schema schema = factory.newSchema(new File("Account.xsd"));
          Validator validator = schema.newValidator();

          validator.validate(new DOMSource(doc));
      } catch (SAXException e) {
          e.printStackTrace();
      }
  }
```

JAX-B

Introduction

Java offers several options for handling XML structures and files. One of the most common and used ones is JAXB. It offers the possibility to convert Java objects into XML structures and the other way around. JAXB comes with the JRE standard bundle since the first versions of the JRE 1.6.

The first specification of JAXB was done in March 2003 and the work process is tracked in the Java Specification Request 31: https://jcp.org/en/jsr/detail?id=31. In this specification request you can find a lot of information regarding the long life of JAXB and all the improvements that have been made.

As already mentioned, JAXB is included in the JRE bundle since the update 1.6. Before that it was necessary to include their libraries in the specific Java project in order to be able to use it.

Before JAXB was available (long time ago), the way Java had to handle XML documents was the DOM: http://www.w3.org/-DOM/. This was not a very good approach because there was almost not abstraction from XML nodes into Java objects and all value types were inferred as Strings. JAXB provides several benefits like Object oriented approach related to XML nodes and attributes, typed values, annotations and may others.

JAXB stands for Java architecture for XML binding. It is used to convert XML to java object and java object to XML. JAXB defines an API for reading and writing Java objects to and from XML documents. Unlike SAX and DOM, we don't need to be aware of XML parsing techniques.

Basically, XML Binding API's are categorized into two types of tools.

Design Time tools – In these type of tools, there will be a Binding compiler which will take a Schema or DTD as input and generated supported classes for converting the XML to Object and Object to XML.

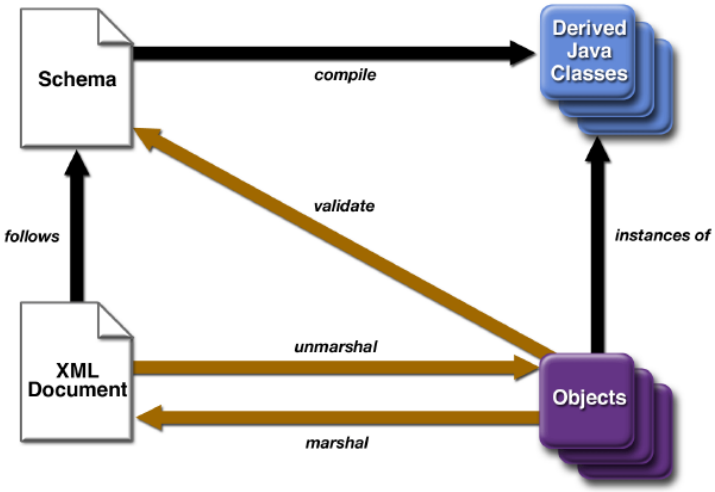
Runtime tools – These will not generate any classes rather on fly will take the XML as an input and based the elements names, matches them with corresponding attribute names of the class and will convert XML into Object of the class.

When it comes to JAX-B API it is a design time tool, where it will come up with its own compiler to generated classes to support the binding process. Before proceeding further on JAX-B let us first try to understand its architecture and its offerings.

Architecture

If a Java Object is following the structure of a Java class then that Object is called Instance of that class. If XML document is following the structure of an XSD document can I call the XML as an Instance of that XSD or not? Certainly, the answer for this would be yes.

In Java, Object structure would be defined by a class. In XML, structure of an XML is defined by an XSD document as shown below.



If we try to derive a conclusion from the above, we can understand that Classes and XSD documents both try to represent the structure, so we can create set of Java classes which represents the structure of an XSD document, then the XML document (holds data) representing the structure of that XSD can be converted into its associated Java class Objects. Even the wise versa would also be possible.

Let us try to deep down into the above point. How to create user defined data type in Java? Using class declaration. How to create user defined data type in XSD? By using ComplexType declaration. Which means a ComplexType in XSD is equal to a Java class in Java. As XSD document means it contains elements and multiple Complex Type's so, to represent the structure of an XSD document in Java, it is nothing but composition of various Java classes (one Java class to one XSD ComplexType) representing its structure.

Mapping or Binding

Java objects can be bonded to XML structures by using certain annotations and following specific rules. This is what we call as mapping or Binding. To perform this binding, we need to generate Binding classes which represents structures of the XML.

Let us take the example of PurchaseOrder XSD to understand better.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
      <xs:element name="purchaseOrder" type="purchaseOrderType" />
      <xs:complexType name="purchaseOrderType">
            <xs:sequence>
                  <xs:element name="orderItems" type="orderItemsType" />
                  <xs:element name="shippingAddress" type="shippingAddressType" />
            </xs:sequence>
      </xs:complexType>
      <xs:complexType name="orderItemsType">
            <xs:sequence>
                  <xs:element name="item" type="itemType" minOccurs="1" maxOccurs="unbounded" />
            </xs:sequence>
      </xs:complexType>
      <xs:complexType name="itemType">
            <xs:sequence>
                  <xs:element name="itemCode" type="xs:string" />
                  <xs:element name="quantity" type="xs:int" />
            </xs:sequence>
      </xs:complexType>
      <xs:complexType name="shippingAddressType">
            <xs:sequence>
                  <xs:element name="addressLine1" type="xs:string" />
                  <xs:element name="addressLine2" type="xs:string" />
                  <xs:element name="city" type="xs:string" />
                  <xs:element name="state" type="xs:string" />
                  <xs:element name="zip" type="xs:int" />
                  <xs:element name="country" type="xs:string" />
            </xs:sequence>
      </xs:complexType>
</xs:schema>
```

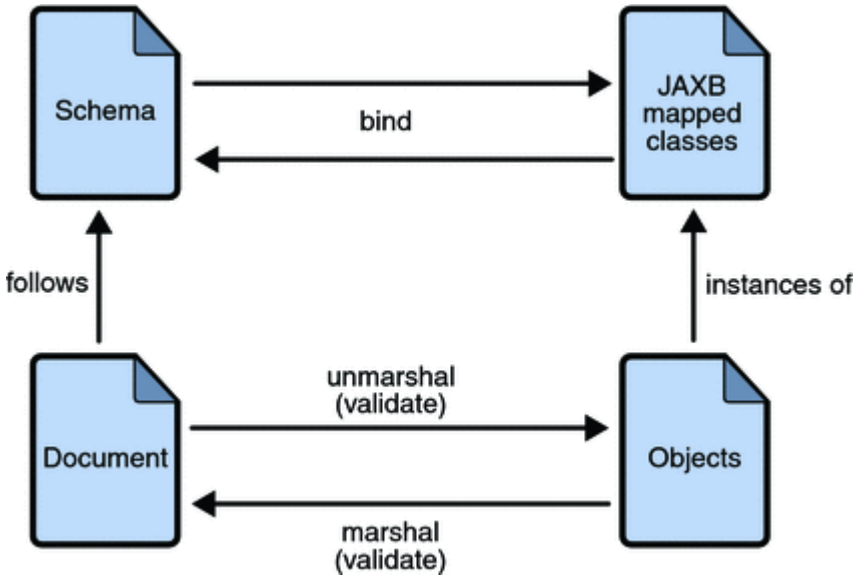
In the above XSD we have one element and four ComplexType declarations, as we discussed earlier ComplexType declarations in XSD are equal to a Java class in Java. So from the above we can derive four Java classes as shown below.

class ShippingAddressType {
 	
StringaddressLine1;
StringaddressLine2;
String city;
String state;
int zip;
String country;
}
	class ItemType {
	
	String itemCode;
	int quantity;
	}

class OrderItemsType {
 	
List<ItemType>item;
}
	class PurchaseOrderType {
	OrderItemsType orderItems;
	ShippingAddressType shippingAddress;
	}

From the above we can create the PurchaseOrderType object representing the root element of XSD, can I hold the PurchaseOrder XML document data into this instance or not? Absolutely the answer would be yes, because PurchaseOrderType is representing the structure of the XSD document, so its instance can hold its XML data.

With this we understood that we can created the Java classes representing the structure of an XSD document, hence those classes are called as Binding Classes as those are bonded to the XSD document structure and the objects out of those classes are called binding objects.



Now we know that we can convert the XML document data into a Binding Class Object, so the process of converting an XML to a Binding class Object is called Un-Marshalling. And the process of converting a Binding Object back into XML representation is called Marshalling and both the types of operations are supported by any of the XML Binding API's including JAX-B.

So, JAX-B supports two types of operations

1) One-Time operations
2) Runtime operations.

Let us discuss them in detail below.

One-Time Operation

One-Time operations are the operations which would be done only once within the lifetime of an XSD or Binding class.

Let's say I want to convert the Purchase Order XML document to Object, this is Un-Marshalling. To perform Un-Marshalling do we need Binding classes representing the structure of the Purchase Order XSD or not? So the programmer has to read the contents of the XSD and based on the number of complex types he needs to create equivalent Java classes representing their structure. As our purchase order XSD document contains only four complex types it would be easy for the developer to determine the relations between the classes and can create. Let's say I have an XSD document in which 100 ComplexType declarations are there, is it easy for a developer to develop 100 Java classes representing the structure of 100 ComplexTypes. Even though it seems to be feasible, but it is a time taking task.

So, instead of writing the Binding classes manually, JAX-B has provided a tool to generated Binding classes from XSD called XJC. And to generate XSD from a Binding class there is another tool called Schemagen.

If I have an XSD document, how many times do I need to generate Java classes to hold the data of its XML? It would be only once, because classes will not hold data they just represents structure, so instances of those classes holds data of an XML. Even the reverse would also be same. As we need to generate the Binding classes

or XSD only once this process is called one-time operations. And there are two types of one-time operations.

a) XJC – XSD to Java compiler – Generates Binding classes from XSD document

b) Schemagen – Schema generator – Generated XSD document from an Binding class

How to use XJC or Schemagen?

Implementing all binding classes for an existing XML interface can be a time consuming and tedious task. But the good news is, you do not need to do it. If you have a XSD schema description, you can use the xjc binding compiler to create the required classes. And even better, xjc is part of the JDK. So there is no need for external tools and you should always have it at hand if required.

Open Command Prompt and type xjc -help command → It gives all xjc options like below

- **-d to define where the generated classes shall be stored in the file system,**
- **-p to define the package to be used and of course**
- **-help if you need anything else.**

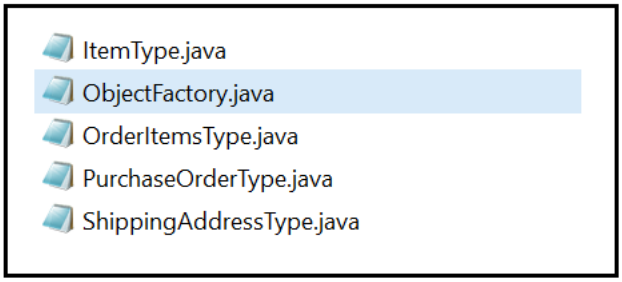
Generating Binding Classes for XSD using XJC

Command Prompt

```
C:\Ashok>xjc -p com.amazon.types Purchase-Order.xsd
parsing a schema...
compiling a schema...
com\amazon\types\ItemType.java
com\amazon\types\ObjectFactory.java
com\amazon\types\OrderItemsType.java
com\amazon\types\PurchaseOrderType.java
com\amazon\types\ShippingAddressType.java

C:\Ashok>
```

<u>What does XJC generates?</u>

When we run XJC.bat by giving XSD as an input along with generating binding classes it generates few other things as explained below.



All the binding classes would be generated into the given package com.amazon.types

1. Package-info.java – XJC after mapping the namespace to a package name, it would store this mapping information into a class called package-info.java. This is also generated under the above mapped package only.

2. ObjectFactory.java – These Class Acts as factory class, which knows how to create objects of all the Binding classes generated. It contains methods like createXXX, upon calling these methods it would return respective binding class objects.

3. Set of Binding classes – With the above it generates Binding classes representing the structure of each ComplexType in XSD. Can we call any java class as Binding class? No, the binding classes are the classes which are bonded to XSD complexType declarations and to denote this relationship, those are marked with JAX-B annotations as described below.

i. @XMLType – To the top of the binding class it would be marked with @XMLType annotation indicated this class is representing an XML ComplexType

ii. @XMLAccessorType(AccessType.FIELD) – In conjunction with @XMLType, one more annotation

@XMLAccessorType(AccessType.FIELD) also would be generated. It denotes how to map XML elements to Binding class attributes. With the AccessType.FIELD we are saying the elements names must be matched with binding class attributes names to port the data.

iii. @XMLElement – This annotation is marked at the attributes of the Binding classes to indicates these attributes are representing elements of XML.

So, by the above we understood that not every Java class is called as Binding class, only the classes which are annotated with JAX-B annotations are called binging classes.

So, from the above binding classes we can even generated XSD document by running the Schemagen tool.

With this we understood how to perform one-time operations. Now we can proceed on runtime operations.

Runtime Operations

Till now we discussed about how to perform one-time operations. Let us try to understand Runtime operations. JAX-B supports three runtime operations.

1) Un-Marshalling – The process of converting XML to Java Object
2) Marshalling – The process of converting Java Object with data to its XML equivalent
3) In-Memory Validation – Before converting an XML document to Java Object first we need to validate whether the XML conforms to XSD or not would be done using In-Memory Validation.

If we want to perform runtime operations, first you need to do one-time operation. Unless you have binding classes or XSD equivalent for any existing binding class, you can't work on runtime operations.

By following the things described in earlier section we assume you already completed the One-Time operations and you have binding classes with you. Let us try to understand how to work on each of the Runtime operations in the following section.

Un-Marshalling:

As described earlier, the process of converting an XML document into Java Object is called Un-Marshalling. In order to perform Un-Marshalling, you need to have Binding Classes with you. You can generate these by running XJC compiler. If you run XJC compiler on the Purchase-Order.xsd which was described earlier, the following binding classes will be generated shown below.

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "ShippingAddressType", propOrder = {
    "addrLine1",
    "addrLine2",
    "city",
    "state",
    "country",
    "zipcode"
})
public class ShippingAddressType {

    @XmlElement(required = true)
    protected String addrLine1;
    @XmlElement(required = true)
    protected String addrLine2;
    @XmlElement(required = true)
    protected String city;
    @XmlElement(required = true)
    protected String state;
    @XmlElement(required = true)
    protected String country;
    protected int zipcode;

   //setters & getter

}
```

ShippingAddressType.java

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "ItemType", propOrder = {
    "itemCode",
    "quantity"
})
public class ItemType {

    @XmlElement(name = "item-code", required = true)
    protected String itemCode;
    protected int quantity;

      //setters & getters

}
```

ItemType.java

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "PurchaseOrderType", propOrder = {
    "orderItems",
    "shippingAddress"
})
public class PurchaseOrderType {

    @XmlElement(name = "order-items", required = true)
    protected OrderItemsType orderItems;
    @XmlElement(name = "shipping-address", required = true)
    protected ShippingAddressType shippingAddress;

      //setters & getters

}
```

PurchaseOrderType.java

```
@XmlRegistry
public class ObjectFactory {

    private final static QName _PurchaseOrder_QNAME = new QName("", "PurchaseOrder");

    public ObjectFactory() { }

    public PurchaseOrderType createPurchaseOrderType() {
        return new PurchaseOrderType();
    }

    public OrderItemsType createOrderItemsType() {
        return new OrderItemsType();
    }

    public ItemType createItemType() {
        return new ItemType();
    }

    public ShippingAddressType createShippingAddressType() {
        return new ShippingAddressType();
    }

    @XmlElementDecl(namespace = "", name = "PurchaseOrder")
    public JAXBElement<PurchaseOrderType> createPurchaseOrder(PurchaseOrderType value) {
        return new JAXBElement<PurchaseOrderType>(_PurchaseOrder_QNAME, PurchaseOrderType.class, null, value);
    }
}
```

ObjectFactory.java

Having the binding classes will not automatically converts the XML document into Object of these classes, rather we need a mediator who would be able to read the contents of the XML, and identify the respective binding class for that and creates objects to populate data. This will be done by a class called "Unmarshaller". As JAX-B is an API, so Unmarshaller is an interface defined by JAX-B API and its JAX-B API implementations (JAX-B RI or JaxMe) contains the implementation class for this interface.

The object of Unmarshaller has a method unmarshal() , this method takes the input as XML document and first check for Well-formness. Once the document is well-formed, its reads the elements of the XML and tries to identify the binding classes for those respective elements based on @XMLType and @XMLElement annotations and converts them into the Objects.

We have two problems here

1) How to create Un-marshaller as we don't know implementation class
2) Where does un-marshaller should search for identifying a binding class (is it in entire JVM memory or classpath?)

As we don't know how to identify the implementation of Unmarshaller interface, we have a factory class called JAXBContext who will finds the implementation class for Unmarshaller interface and would instantiate. It has a method createUnmarshaller() which returns the object of unmarshaller. **This addressed the concern #1**

While creating the JAXBContext, we need to supply any of the three inputs stated below.

1) Package name containing the binding classes
2) ObjectFactory.class reference as it knows the binding classes.
3) Individual Binding classes as inputs to the JAXBContext.

```
JAXBContext jaxbContext=JAXBContext.newInstance("packagename" or
"ObjectFactory.class" or "BC1, BC2, BC3…");
```

With the above statement, the JAXBContext would be created and loads the classes that are specified in the package or referred by ObjectFactory or the individual binding classes specified and creates an in-memory context holding these.

Now when we call the Unmarshaller.unmarshal() method, the Unmarshaller will searches with the classes loaded in the JAXBContext and tries to identify the classes for an XML element based on their annotations. The code for UnMarshalling has shown below.

```
package com.po.unmarshall;

public class POUnmarshaller {
      public static void main(String args[]) throws SAXException {
            try {
                  JAXBContext jContext = JAXBContext.newInstance("com.amazon.types");
                  Unmarshaller poUnmarshaller = jContext.createUnmarshaller();
                  JAXBElement<PurchaseOrderType> jElement = (JAXBElement<PurchaseOrderType>) poUnmarshaller
                              .unmarshal(new File("D:\\Resources\\po.xml"));
                  PurchaseOrderType poType = jElement.getValue();

                  System.out.println(poType.getShippingAddress().getAddressLine1());
                  System.out.println(poType.getOrderItems().getItem().get(0).getItemCode());
            } catch (JAXBException e) {
                  e.printStackTrace();
            }
      }
}
```
POUnmarshaller.java

Marshalling:

Marshalling is the process of converting the Object data to XML. For performing Marshalling also we need to create the Binding classes first. Once we have the Binding classes, we need to create the Object of these classes and should populate data in them.

These objects can be marshaled to XML using the Marshaller. Again, Marshaller is an interface, to instantiate it we need JAXBContext. Then we need to call a method marshal by giving input as XML and Outputstream where we want to generate the output to. The below program depicts the same.

```
package com.po.unmarshall;

public class POMarshaller {
      public static void main(String args[]) {
            try {
                  JAXBContext jContext = JAXBContext.newInstance("com.amazon.types");
                  Marshaller poMarshaller = jContext.createMarshaller();

                  ShippingAddressType shippingAddress = new ShippingAddressType();
                  shippingAddress.setAddressLine1("CDAC Block");
                  shippingAddress.setAddressLine2("S.R.Nagar");
                  shippingAddress.setCity("Hyd"); shippingAddress.setState("AP");
                  shippingAddress.setCountry("India"); shippingAddress.setZip(24244);

                  ItemType item1 = new ItemType();
                  item1.setItemCode("IC2002"); item1.setQuantity(34);

                  OrderItemsType oit = new OrderItemsType();
                  oit.getItem().add(item1);
                  PurchaseOrderType pot = new PurchaseOrderType();
                  pot.setOrderItems(oit);
                  pot.setShippingAddress(shippingAddress);

                  poMarshaller.marshal(pot, System.out);
            } catch (JAXBException e) {
                  e.printStackTrace();
            }
      }
}
```
POMarshaller.java

In-Memory Validation

While performing the Unmarshalling or Marshalling, we need to first of all validate whether the XML is valid against the XSD document or not, and we will validate using the In-Memory validation. An XML can be validated against an XSD document, so we need to read the XSD document and should represent it in a Java Object called Schema.

But Schema is an interface and to create the Object of interface, we need factory called Schema Factory. But schema factory can created various types of XSD documents, so we need to provide the grammar of XSD document which you want to load or created and would be done as show below.

```
SchemaFactory sFactory =
SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_URI);
```

With the above we created a SchemaFactory which is capable of creating or loading W3C XML Schema type documents.

Now we need to load the XSD document into the Schema class object as shown below.

```
Schema poSchema = sFactory.newSchema(new File("Purchase-Order.xsd"));
```

Once the schema object is created holding the structure of XSD, we need to give it to Unmarshaller before calling the unmarshal(). So, that Unmarshaller will validate the input XML with this supplied Schema Object and then performs Unmarshalling. Otherwise Unmarshaller will throw exception, indicating the type of validation failure. The below programs shows how to perform in-memory validation.

```
package com.po.unmarshall;

public class POUnmarshaller {
      public static void main(String args[]) throws SAXException {
            SchemaFactory sFactory = SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
            Schema poSchema = sFactory.newSchema(new File("Purchase-Order.xsd"));

            JAXBContext jContext = JAXBContext.newInstance("com.amazon.types");
            Unmarshaller poUnmarshaller = jContext.createUnmarshaller();

            poUnmarshaller.setSchema(poSchema);

            JAXBElement<PurchaseOrderType> jElement =
            (JAXBElement<PurchaseOrderType>) poUnmarshaller.unmarshal(new File("po.xml"));

            PurchaseOrderType poType = jElement.getValue();

            System.out.println(poType.getShippingAddress().getAddressLine1());
            System.out.println(poType.getOrderItems().getItem().get(0).getItemCode());
      }
}
```

POUnmarshaller.java

Getting Started with Webservices

Before discussing Web Service development, first we need to understand certain aspects related to Web Services. The points we are discussing below are common for both JAX-RPC and JAX-WS as well.

Types of Web Services

As discussed earlier, we have two types of Web Services JAX-RPC API adhering the BP 1.0, and JAX-WS API adhering to BP 1.1. Now while developing a Web service, first of we need to choose a type. After choosing the type, we need to select an appropriate implementation and various other aspects described in following sections.

Web Service Development parts

Always a Web Service development involves two parts. Provider and Consumer. Provider is the actual Web service, who will handle the incoming requests from the consumer. The Consumer is the Client Program which we develop to access the Provider. Now both JAX-RPC and JAX-WS API's has provided classes and interfaces to develop Consumer as well as Provider programs.

Ways of developing a Web Service

There are two ways of developing a Web service

Contract First approach: In Web Services, WSDL acts as a contract between Consumer and Provider. If consumer wants to know the information about the provider, he needs to see the WSDL document. In a Contract First approach, the development of the provider will starts with WSDL, from which the necessary Java classes required to build the service would be generated. As the development starts from WSDL, hence it is called Contract First approach.

Contract Last approach: Here the developer first starts developing the service by writing the Java code, once done then he will generate the WSDL contract to expose it as a Web Service. As the contract is being generated at the end after the development is done, this is called Contract Last approach.

Choosing an Endpoint

In Web services the provider is often referred as Endpoint, as he is the ultimate point from whom we request data or information.

Now in a typical Web Service communication, the consumer will send the initial request with some data to the Provider. The consumer cannot send Object data over the network; rather he needs to convert it into XML and needs to place the XML into SOAP XML. Then the SOAP XML is being transported over the HTTP protocol. This means the consumer will send a HTTP request, to receive the HTTP request being sent by the consumer, at the Provider end we need to have a HTTP Listener. The listener will listen for the incoming request and forwards it for further processing as shown below So we understood we need a listener at the provider side to handle the incoming request. Do we have any existing API's in Java which are capable of handling incoming HTTP Requests? Those are nothing but Servlet or EJB.

There are two types of Endpoints, where a developer can choose while developing a Provider.

Servlet Endpoint: Here at the provider end, the servlet acts as a listener in listening for the incoming request from the client and would process.

EJB Endpoint: In place of Servlet even we can substitute it with EJB, and we can make EJB's to handle even HTTP Request as well.

Most of the people prefer to use Servlet endpoint rather than EJB endpoint as EJB seems to be heavy and costly component, and for the sake of developing Web services, the application needs to be hosted on an EJB container.

Message Exchange Patterns

Message exchange format talks about how a consumer does exchanges information with the provider. This could happen in three ways as described below.

Synchronous request/reply: In this pattern, the consumer sends the request with some data and will waits till the provider finishes processing request and returns the response to the consumer.

Class A is calling a method execute () on B, until the method execute () finishes execution and returns value, the do() method will be blocked on the execute() method call, as it is blocked on method call, it is called blocking request/reply or synchronous request reply. In this case also the Consumer program would be blocked on the Providers method call (request) and will not proceeds execution until the Provider returns response. Which means the Consumer opened HTTP Connection would be open till the Provider returns response.

Asynchronous request/reply or Delayed Response: In this pattern, the Consumer will not be blocked on the Providers method call/request, rather after Consumer sending the request; he disconnects/closes the HTTP Connection. The provider upon receiving the request will hold it and process it after certain amount of time. Once the response is ready will opens a new connection back to the Consumer and dispatches the response.

For e.g. let us consider an example like you are sending a letter to your friend, when you post the letter it would be carried by the postmaster to your friend. Once the letter has been received, your friend may not immediately read the letter, rather when he finds leisure he will read. Till your friend reads the letter and give the reply the postmaster can't wait at your friend's door rather he moves on. Now when your friend reads your letter, then he will reply to your letter by identifying the from address on the letter and post back's you the reply. As the response here is not immediate and is being sent after some time, this is also called as Delayed response pattern.

One way Invoke or Fire and forget: In this case, the consumer will send the request with some data, but never expects a response from the provider. As the communication happens in only one direction it is called One-Way invoke as well.

Message Exchange formats

This describes about how the consumer will prepares/formats the xml message that should be exchanged with the provider.

For e.g. a lecturer is dictating the notes, every student has their own way of writing the notes. Someone may right the notes from bottom to top, others might right from top to bottom. In the same way, to send information, the consumer has to prepare his data in XML format. How does consumer represents the data in the XML document is called Message Exchange format?

The message exchange format would be derived by two parts

1) Style and other is 2) Use.

a) Style – The possible values the Style part can carry is rpc and document

b) Use – This could be encoded and literal

With the above it results in four combinations as rpc-encoded, rpc-literal, document-literal and document-encoded. Out of which document-encoded is not supported by any web service specification. We will discuss the remaining three in later sections.

JAX-WS

Introduction

Java API for XML Web Services (JAX-WS), is a standardized API for creating web services in XML format (SOAP). JAX-WS provides support for SOAP (Simple Object Access protocol) based web services.

Web services have been around a while now. First there was SOAP. But SOAP only described what the messages looked like. Then there was WSDL. But WSDL didn't tell you how to write web services in Java. Then along came JAX-RPC 1.0. After a few months of use, the Java Community Process (JCP) folks who wrote that specification realized that it needed a few tweaks, so released JAX-RPC 1.1. After a year or so of using that specification, the JCP folks wanted to build a better version: JAX-RPC 2.0. A primary goal was to align with industry direction, but the industry was not merely doing RPC web services, they were also doing message-oriented web services. So "RPC" was removed from the name and replaced with "WS" (which stands for web Services). Thus the successor to JAX-RPC 1.1 is JAX-WS 2.0 - the Java API for XML-based web services.

What remains the same in JAX-RPC and JAX-WS?

Before we talk the differences between JAX-RPC 1.1 and JAX-WS 2.0, we should first discuss similarities between JAX-RPC and JAX-WS

- JAX-WS still supports SOAP 1.1 over HTTP 1.1, so interoperability will not be affected. The same messages can still flow across the wire.
- JAX-WS still supports WSDL 1.1, so what you've learned about that specification is still useful. A WSDL 2.0 specification is nearing completion, but it was still in the works at the time that JAX-WS 2.0 was finalized.

What is different between JAX-RPC and JAX-WS?

SOAP 1.2: JAX-RPC and JAX-WS support SOAP 1.1. JAX-WS also supports SOAP 1.2.

XML/HTTP: The WSDL 1.1 specification defined an HTTP binding, which is a means by which you can send XML messages over HTTP without SOAP. JAX-RPC ignored the HTTP binding. JAX-WS adds support for it.

WS-I's Basic Profiles: JAX-RPC supports WS-I's Basic Profile (BP) version 1.0. JAX-WS supports BP 1.1. (WS-I is the web services interoperability organization.)

New Java features: JAX-RPC maps to Java 1.4. JAX-WS maps to Java 5.0. JAX-WS relies on many of the features new in Java 5.0.

Java EE 5, the successor to J2EE 1.4, adds support for JAX-WS, but it also retains support for JAX-RPC, which could be confusing to today's web services novices.

The data mapping model:

- JAX-RPC has its own data mapping model, which covers about 90 percent of all schema types. Those that it does not cover are mapped to javax.xml. soap.SOAPElement.
- JAX-WS's data mapping model is JAXB. JAXB promises mappings for all XML schemas.

The interface mapping model:

- JAX-WS's basic interface mapping model is not extensively different from JAX-RPC's; however:
- JAX-WS's model makes use of new Java 5.0 features.
- JAX-WS's model introduces asynchronous functionality.

The dynamic programming model:

JAX-WS's dynamic client model is quite different from JAX-RPC's. Many of the changes acknowledge industry needs:

- It introduces message-oriented functionality.
- It introduces dynamic asynchronous functionality.
- JAX-WS also adds a dynamic server model, which JAX-RPC does not have.
- MTOM (Message Transmission Optimization Mechanism)
- JAX-WS, via JAXB, adds support for MTOM, the new attachment specification. Microsoft never bought into the SOAP with Attachments specification; but it appears that everyone supports MTOM, so attachment interoperability should become a reality.

The handler model:

The handler model has changed quite a bit from JAX-RPC to JAX-WS.

- JAX-RPC handlers rely on SAAJ 1.2. JAX-WS handlers rely on the new SAAJ 1.3 specification.

SOAP 1.2

There is really not a lot of difference, from a programming model point of view, between SOAP 1.1 And SOAP 1.2. As a Java programmer, the only place you will encounter these differences is
When using the handlers.

XML/HTTP

Like the changes for SOAP 1.2, there is really not a lot of difference, from a programming model Point of view, between SOAP/HTTP and XML/HTTP messages. As a Java programmer, the only Place you will encounter these differences is when using the handlers. The HTTP binding has its own handler chain and its own set of message context Properties.

WS-I's basic profiles

JAX-RPC 1.1 supports WS-I's Basic Profile (BP) 1.0. Since that time, the WS-I folks have Developed BP 1.1 (and the associated AP 1.0 and SSBP 1.0). These new profiles clarify some Minor points, and more clearly define attachments. JAX-WS 2.0 supports these newer profiles. For the most part, the differences between them do not affect the Java programming model. The Exception is attachments. WS-I not only cleared up some questions about attachments, but they Also defined their own XML attachment type: wsi:swaRef. Many people are confused by all these profiles. You will need a little history to clear up the confusion.

WS-I's first basic profile (BP 1.0) did a good job of clarifying the various specs. But it wasn't perfect. And support for SOAP with Attachments (Sw/A) in particular was still rather fuzzy. In their second iteration, the WS-I folks pulled attachments out of the basic profile - BP 1.1 - and fixed some of the things they missed the first time around. At that point they also added two mutually exclusive supplements to the basic profile: AP 1.0 and SSBP 1.0. AP 1.0 is the Attachment Profile which describes how to use Sw/A. SSBP 1.0 is the Simple SOAP Binding Profile, which

describes a web services engine that does not support Sw/A (such as Microsoft's .NET). The remaining profiles that WS-I has and is working on build on top of those basic profiles.

Summary

JAX-WS 2.0 is the successor to JAX-RPC 1.1. There are some things that haven't changed, but most of the programming model is different to a greater or lesser degree. The topics introduced in this tip will be expanded upon in a series of tips which we will publish over the coming months that will compare, in detail, JAX-WS and JAX-RPC. At a high level though, here are a few reasons why you would or would not want to move to JAX-WS from JAX-RPC.

Reasons to stay with JAX-RPC 1.1:

- If we want to stay with something that's been around a while, JAX-RPC will continue to be supported for some time to come.
- If we don't want to step up to Java 5.
- If we want to send SOAP encoded messages or create RPC/encoded style WSDL.

Reasons to step up to JAX-WS 2.0:

- If we want to use the new message-oriented APIs.
- If we want to use MTOM to send attachment data.
- If we want better support for XML schema through JAXB.
- If we want to use an asynchronous programming model in your web service clients.
- If we need to have clients or services that can handle SOAP 1.2 messages.
- If we want to eliminate the need for SOAP in your web services and just use the XML/HTTP binding.

SOAP

SOAP is an XML specification for sending messages over a network. SOAP messages are independent of any operating system and can use a variety of communication protocols including HTTP and SMTP.

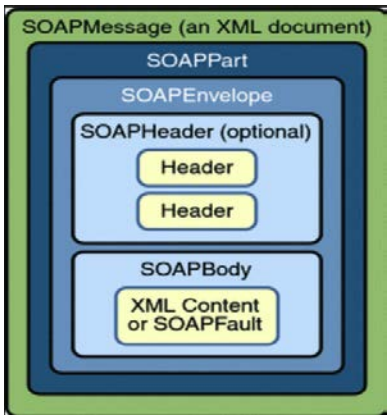
SOAP is XML heavy, hence best used with tools/frameworks. JAX-WS is a framework that simplifies using SOAP. It is part of standard Java.

SOAP message contains the following three parts:

Envelope: This Envelope element contains an optional Header element and a mandatory Body element.

Header: SOAP message contains the information which explains how the message is to be processed.

Body: Body part of the SOAP message contains the actual payload of the end to end message transmission.



Java-XML Data Binding

SOAP-based web services use XML to exchange request and response messages. This requires an architecture for converting Java objects to XML and the reverse. JAXB (Java Architecture for XML Binding) was developed for this purpose.

JAX-RPC uses its own data mapping model. This is because the JAXB specification had not been finalized when the first version of JAX-RPC was completed. The JAX-RPC data mapping model lacks support for some XML schemas.

JAX-WS uses JAXB for data binding. JAXB provides mapping for virtually all schemas.

We can use JAXB annotations on your Java bean and JAX-WS will convert it and its properties to XML elements at runtime when sending the SOAP message.

```
import javax.xml.bind.annotation.*;

@XmlRootElement(name = "person")
@XmlType(propOrder = {"id","name"})
public class Person {

    @XmlElement(name = "id", required = true)
    int id;
    @XmlElement(name = "name", required = true)
    String name;
    // accessors and mutators
}
```

Web Services Definition Language (WSDL)

The Web Services Description Languages is an XML-based interface definition language that is used for describing the functionality offered by a web service. The acronym is also used for any specific WSDL description of a web service (also referred to as a WSDL file), which provides a machine-readable description of how the service can be called, what parameters it expects, and what data structures it returns. Therefore, its purpose is roughly similar to that of a method signature in a programming language.

WSDL 1.0 (Sept. 2000) was developed by IBM, Microsoft, and Ariba to describe Web Services for their SOAP toolkit. It was built by combining two service description languages: NASSL (Network Application Service Specification Language) from IBM and SDL (Service Description Language) from Microsoft.

WSDL 1.1, published in March 2001, is the formalization of WSDL 1.0. No major changes were introduced between 1.0 and 1.1.

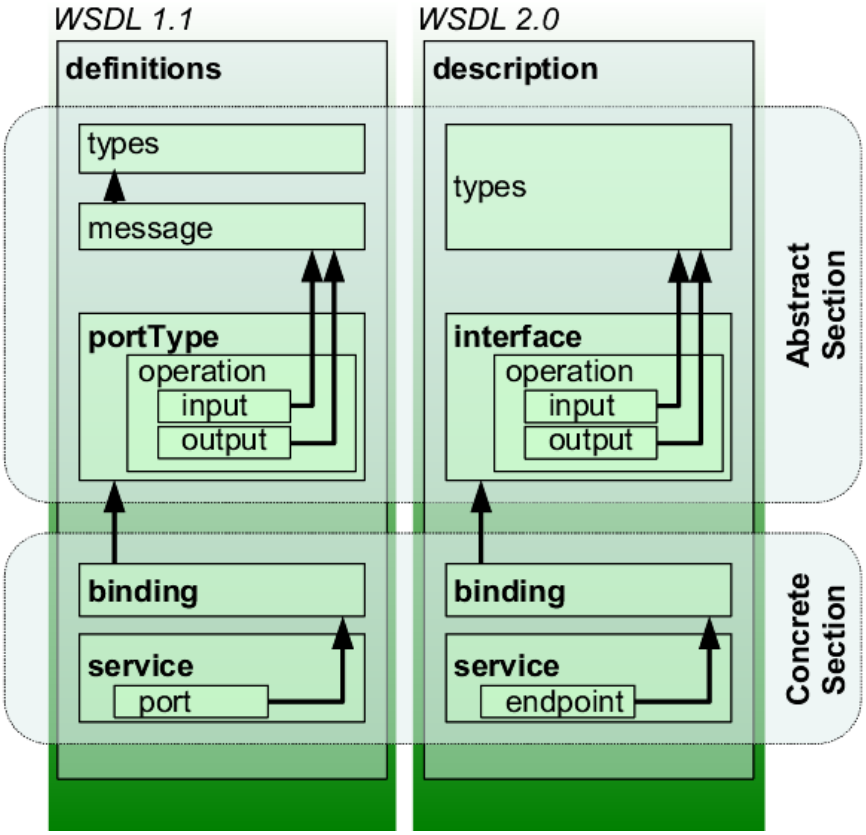
WSDL 1.2 (June 2003) was a working draft at W3C, but has become WSDL 2.0. According to W3C: WSDL 1.2 is easier and more flexible for developers than the previous version. WSDL 1.2 attempts to remove non-interoperable features and also defines the HTTP 1.1 binding better. WSDL 1.2 was not supported by most SOAP servers/vendors.

WSDL 2.0 became a W3C recommendation on June 2007. WSDL 1.2 was renamed to WSDL 2.0 because it has substantial differences from WSDL 1.1. The changes are the following:

Added further semantics to the description language

- Removed message constructs
- Operator overloading not supported
- PortTypes renamed to interfaces
- Ports renamed to endpoints

The current version of WSDL is WSDL 2.0.



Definition: The definition element must be the root element of the WSDL specification.

This element defines the name of the web service, information about the multiple namespaces used though out the WSDL document.

Types: This element is very important while sharing the data across multiple platforms using web services. The data type of the variables we are using in the web services should be compatible with all platforms. The types element is used to define all the data types used between the end to end transmissions. XSD or XML Schema Definition is used to define the types used in the WSDL. The detailed overview of XSD will explain in the next section.

Message: Each message describes either the input or output of the web service transmission and it has the name of the message and zero or more-part message elements Input message contains the request information and the corresponding part message details. The output message element contains the details about the response from the server.

portType: PortType elements are an entity like a Java class containing group of operations.

This element describes a complete operation definition by combining input and output messages. One portType element can describe multiple operations

Binding: The binding element describes how the service will be implemented in the transmission.

Service: Service element provides the address, information about the service. This element defines the URL of the service we are invoking

Top-Down vs. Bottom-Up

There are two ways of building SOAP web services. We can go with a top-down approach or a bottom-up approach.

In a top-down (contract-first) approach, a WSDL document is created, and the necessary Java classes are generated from the WSDL. In a bottom-up (contract-last) approach, the Java classes are written, and the WSDL is generated from the WSDL.

Writing a WSDL file can be quite difficult depending on how complex your web service is. This makes the bottom-up approach an easier option. On the other hand, since your WSDL is generated from the Java classes, any change in code might cause a change in the WSDL. This is not the case for the top-down approach.

Features of JAX-WS

- Requests and responses are transmitted as SOAP messages (XML files) over HTTP
- Even though SOAP structure is complex, developer need not bother about the complexity in messaging. The JAX-WS run time system manages all the parsing related stuffs.
- Since JAX-WS uses the W3C defined technologies, a SOAP web service client can access a SOAP web service that is running in a non-java platform.
- A machine readable WSDL (Web Service Description Language) file which describes all the operations about a service can be present in case of SOAP web services
- Annotations can be used with JAX-WS to simplify the development

Annotations used in JAX WS

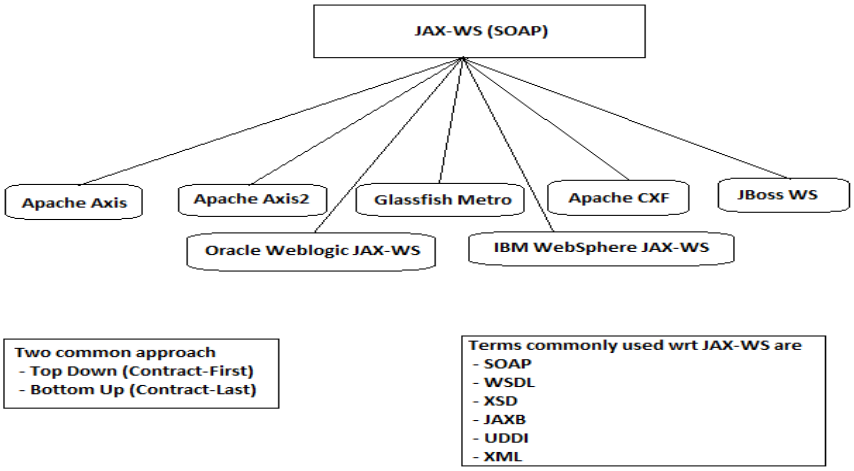
Annotations can be used in Java classes in creating web services. The most commonly used annotations are listed below:

- **@WebService**
- **@WebMethod**
- **@WebParam**
- **@WebResult**
- **@SOAPBinding**
- **@OneWay**

As JAX-WS is an API we can't start the development directly with API. We need an implementation for this API.

JAX-WS implementations

There are several implementations for JAX-WS. Some of them are:



Developing WebService using JAX-WS API with RI implementation

Deciding Technology Stack

Specification : B.P 1.1

API : JAX-WS

Implementation : Reference Implementation (RI)

Role : Provider

Approach : Contract Last Approach

Endpoint : Servlet Endpoint

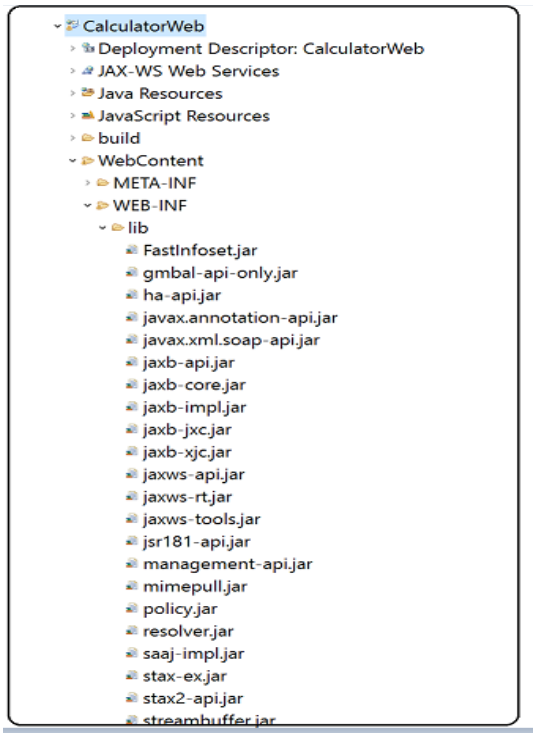
MEP **:** Synchronous request - reply

MEF **:** Document Literal (By default)

Requirement: Develop an application to perform Airthematic Operations through WebService

Procedure of developing Provider

Step 1: - Create Dynamic web project in IDE and add jax-ws RI related jars in project lib folder



Step 2: - Create Service Endpoint Interface (This is optional in jax-ws api)

ICalculator.java

```java
package com.cal.service;

import javax.jws.WebMethod;
import javax.jws.WebService;

@WebService(name = "ICalculator")
public interface ICalculator {

    @WebMethod(operationName = "Add")
    public Integer add(Integer a, Integer b);

    @WebMethod(operationName = "Subtract")
    public Integer subtract(Integer a, Integer b);

}
```

Step3: - Create Service Implementation class

Step 4: - Create Web service deployment descriptor file in project WEB-INF folder (sun-jaxws.xml) with endpoint and url-pattern details

Step 5: Configure WSServletContextListener & WSServlet in web.xml

Step 6: Deploy the application and access using Endpoint URL

Access the WSDL using below URL

Note: If we are able access the WSDL like above, that means Provider is up and running

Step 7: Test the Provider using SOAP UI tool

Open SOAP UI - > Click on File - > New SOAP Project -> Enter Project Name and WSDL URL

Step 8: Send SOAP request like below - > we can SOAP Response

Developing Consumer

After developing the provider, to access provider, we need to build the Consumer.

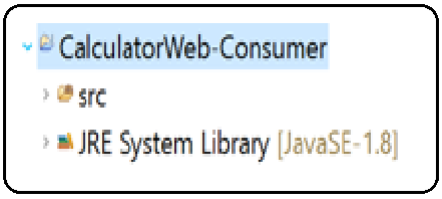
JAX-WS API supports both the developments of Provider as well as Consumer. JAX-WS API supports two types of consumer, they are

- **Stub based consumer**
- **Dispatch API**

From above two approaches, most of the projects use stub based consumer and following section describes how to build a stub based consumer for JAX-WS Provider.

Procedure for Developing STUB Based Consumer for JAX-WS Provider

Step 1: Open IDE and create a Java Project

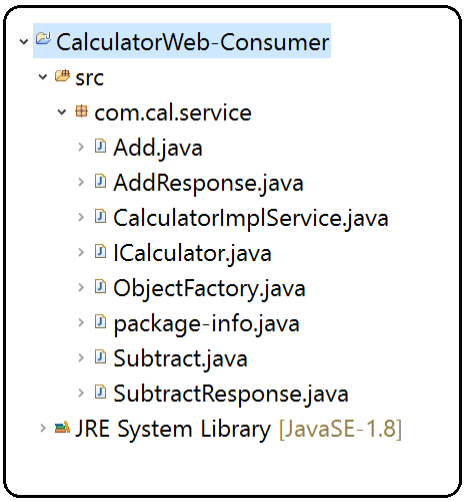


Step 2: Generate classes required for consumer using wsimport tool by giving wsdl URL as input

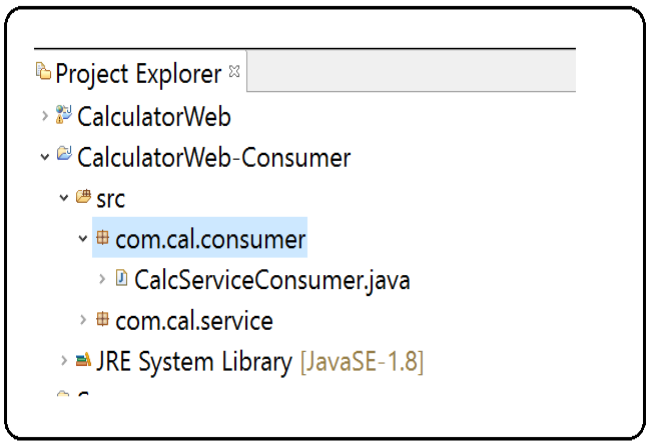
Syntax: wsimport –d src –keep –verbose WSDL_URL

Note: wsimport tool will be shipped as part of JDK itself

After classes are generated -> Refresh the project - > we can see classes like below



Step 3: Develop Consumer class to access the provider



Step 4: Run CalcServiceConsumer.java class

Contract First Approach

Step1: Create Dynamic web project and add jar files in project lib folder

Step 2: Create WSDL file in project WEB-INF folder

----------WeatherService.wsdl Start----------

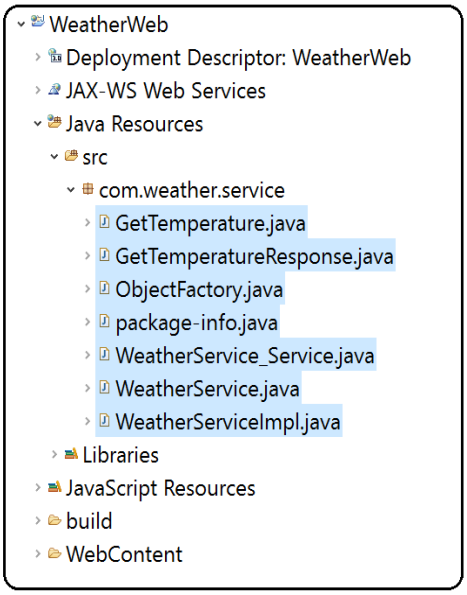
```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://service.weather.com"
       xmlns:apachesoap="http://xml.apache.org/xml-soap"
xmlns:impl="http://service.weather.com"
       xmlns:intf="http://service.weather.com" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
       xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
       <!--WSDL created by Apache Axis version: 1.4 Built on Apr 22, 2006 (06:55:48  PDT) -->
       <wsdl:types>
              <schema elementFormDefault="qualified"
targetNamespace="http://service.weather.com"
                     xmlns="http://www.w3.org/2001/XMLSchema">
                     <element name="getTemperature">
                            <complexType>
                                   <sequence>
                                          <element name="zip" type="xsd:int" />
                                   </sequence>
                            </complexType>
                     </element>
                     <element name="getTemperatureResponse">
                            <complexType>
                                   <sequence>
                                          <element name="getTemperatureReturn"
type="xsd:double" />
                                   </sequence>
                            </complexType>
                     </element>
              </schema>
       </wsdl:types>

       <wsdl:message name="getTemperatureRequest">
              <wsdl:part element="impl:getTemperature" name="parameters">
              </wsdl:part>
       </wsdl:message>
       <wsdl:message name="getTemperatureResponse">
              <wsdl:part element="impl:getTemperatureResponse" name="parameters">
              </wsdl:part>
       </wsdl:message>
       <wsdl:portType name="WeatherService">
              <wsdl:operation name="getTemperature">
                     <wsdl:input message="impl:getTemperatureRequest"
name="getTemperatureRequest">
                     </wsdl:input>
                     <wsdl:output message="impl:getTemperatureResponse"
name="getTemperatureResponse">
```

```
                    </wsdl:output>
            </wsdl:operation>

        </wsdl:portType>
        <wsdl:binding name="WeatherServiceSoapBinding" type="impl:WeatherService">
            <wsdlsoap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http" />
            <wsdl:operation name="getTemperature">
                <wsdlsoap:operation soapAction="" />
                <wsdl:input name="getTemperatureRequest">
                    <wsdlsoap:body use="literal" />
                </wsdl:input>
                <wsdl:output name="getTemperatureResponse">
                    <wsdlsoap:body use="literal" />
                </wsdl:output>
            </wsdl:operation>
        </wsdl:binding>

        <wsdl:service name="WeatherService">
            <wsdl:port binding="impl:WeatherServiceSoapBinding" name="WeatherService">
                <wsdlsoap:address

        location="http://localhost:4040/WeatherWeb/services/WeatherServiceImpl" />
            </wsdl:port>
        </wsdl:service>
</wsdl:definitions>
```

----------WeatherService WSDL End----------

Step-3: Generate Binding classes based on WSDL file using wsimport tool

Syntax: wsimport –d src –keep –verbose PATH_OF_WSDL



Step 4: Create WeatherServiceImpl.java with business logic

Step 5 : Create WebService deployment descriptor file in project WEB-INF folder (sun-jaxws.xml)

Step 6: Configure WSServlet in web.xml file

Step 7: Deploy the application and access the Endpoint

Step 8: Test the Provider using SOAP UI tool

SOAP Handlers

- Handlers are the message interceptors which does additional processing of the inbound and outbound messages.
- Inbound messages are the request messages (client to web service)
- Outbound messages are the response messages (web service to client)
- Additional processing of a SOAP message would be to add some additional information to SOAP headers or manipulate the values based on some condition etc.

Usecase-1:

- Attach most commonly used parameters in the SOAP header using a handler at the client side.
- In the server side, write a handler which can check these commonly used parameters from the header and then produce the response from the cache if these parameters are found otherwise get the data from the backend.
- This way it will improve the web service performance.

Usecase-2:

- In the client-side handler, add some client identity information in the SOAP header like IP address of a client
- In the server-side handler, extract the IP address from the SOAP header and check whether it belongs to trusted IP addresses or not and then process it further if it's a trusted IP else throw an exception which can be sent as SOAP fault

JAX-WS provides 2 types of Message handlers

1. SOAP handlers
2. Logical handlers

SOAP handlers: SOAP handlers are the protocol specific handlers and they can access or change the protocol specific aspect (like headers) of a message and also, they can alter the body of a SOAP message

Basically, SOAP handlers can access the entire SOAP message (header + body)

Logical handlers: Logical handlers are protocol-agnostic and cannot access or change any protocol-specific parts (like headers) of a message.

They act only on the payload of the message.

Basically, Logical handlers can access only body of the message.

Note: The most preferred handler is the SOAP handler as it provides an access to the entire SOAP message (header + body)

Message Context

Message context provides methods to access and modify the inbound and outbound messages.

It also provides some properties which can be used to determine whether message is inbound or outbound.

- **Each SOAP Handler should implement javax.xml.ws.handler.soap.SOAPHandler interface**
- **Each Logical handlers should implement javax.xml.ws.handler.LogicalHandler interface**
- **Message context for logical handlers is represented by javax.xml.ws.handler.LogicalMessageContext**
- **Message context for SOAP handlers is represented by javax.xml.ws.handler.soap.SOAPMessageContext**
- **Message for logical handler is represented by javax.xml.ws.LogicalMessage**
- **Message for SOAP handler is represented by javax.xml.soap.SOAPMessage**

We can write both SOAP handler and Logic handler within a handler chain but in this case, logical handlers will be executed before the SOAP handlers on outbound message and SOAP handlers are executed before the logic handlers on inbound message. We can see that in below diagram

How we define handlers?

To define a handler we just need to implement the handler interface and override the below 3 methods

1.handleMessage()

2.handleFault()

3.close()

handleMessage ( ) : This method will be called for both inbound and outbound messages Perform the additional processing of the message in this method.

handleFault () : This method handles the processing of any SOAP faults generated by the handleMessage() method, and also the faults generated by the back-end component

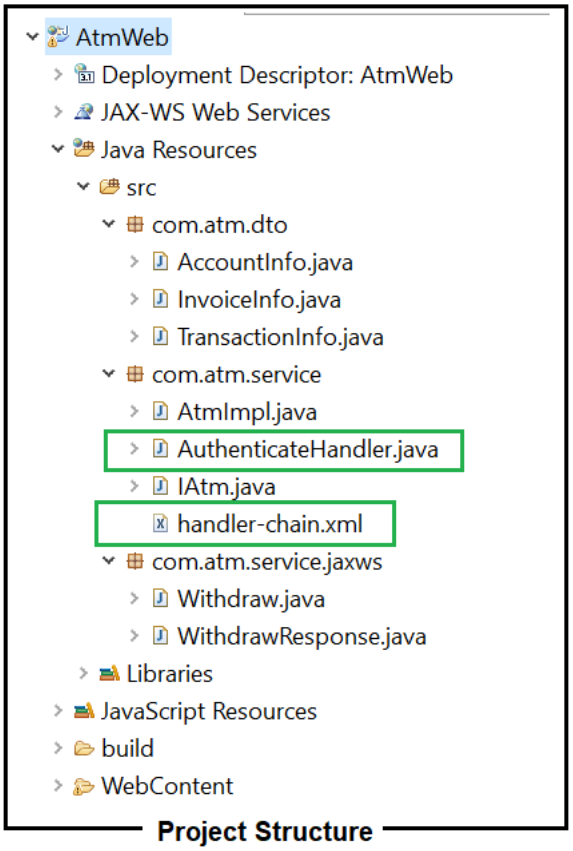
close () : This is called after the completion of message processing by all handlers for each web service invocation. This method can be used to clean up all resources which are used during processing of the message.

Webservice development using SOAP Handler.

We are going to develop a Webservice provider i.e. AtmWeb. This Provider(IAtm.java) contains a withdraw(..) method to withdraw method. When consumer send request for withdraw(..) method, first I want to validate consumer using a Secret Key. To validate consumer sent secret key I am developing Authentication Handler (SOAP Handler). If secret-key is matched then consumer request will be processed and returns the successful response if secret-key is not matched then Provider will return SOAP Fault Exception as a response to Consumer.

As part of this Authentication Handler I am using Secret Key as → **"Ashok-IT"**

Step-1: - Create a project like below



<u>Provider contains below classes</u>

AccountInfo.java, InvoiceInfo.java and TransactionInfo.java → Model classes

IAtm.java → Service Endpoint interface

AtmImpl.java → Service Endpoint Interface Implementation class

AuthenticationHandler.java → AuthenticationHandler to validate Secret Key

Handler-chain.xml → Handler Chain configuration file

Withdraw.java and WithdrawResponse.java → Binding classes

Below are the model classes

```
public class AccountInfo {

	private String accId;
	private String name;
	private String branch;

	//setters & getters
}
```
AccountInfo.java

```
public class InvoiceInfo {

	private String invoiceNum;
	private String invoiceGenDate;
	private String status;
	private String denialRsn;
	private String txNo;

	//setters & getters
}
```
InvoiceInfo.java

```
public class TransactionInfo {

	private String atmId;
	private String pin;
	private String amount;

	//setters & getters
}
```
TransactionInfo.java

Below are the Service Endpoint interface and Impl classes

----------AuthenticateHandler.java start----------

```
public class AuthenticateHandler implements SOAPHandler<SOAPMessageContext> {

	//Choosing some secret key for validation
	private static final String SECRET_KEY = "Ashok-IT";

	@Override
	public void close(MessageContext ctx) {
	}
	@Override
	public boolean handleFault(SOAPMessageContext arg0) {
		// TODO Auto-generated method stub
		return false;
	}
	/**
	 * This method is used to handle inbound and outbound soap msgs
	 */
	@Override
	public boolean handleMessage(SOAPMessageContext ctx) {
	 Boolean outbound = (Boolean) ctx.get(MessageContext.MESSAGE_OUTBOUND_PROPERTY);
		if (!outbound) {
			SOAPMessage soapMsg = ctx.getMessage();
			try {
				soapMsg.writeTo(System.out);
				// Grabbing soap header from soap msg
				SOAPHeader soapHeader = soapMsg.getSOAPHeader();
	Iterator<?> headerIterator =
	soapHeader.extractHeaderElements(SOAPConstants.URI_SOAP_ACTOR_NEXT);
				// If soap header presented then process
				if (headerIterator.hasNext()) {
					Node node = (Node) headerIterator.next();
```

```
                    if (node != null) {
                        String secretKey = node.getValue();
                        if (SECRET_KEY.equals(secretKey.trim())) {
                            // Key matching.. process the request
                            return true;
                        } else {
                            // Key did not match
                            // Throw SOAP Fault exception
                            generateSoapFaultException(soapMsg);
                        }
                    }
                } else {
                    // Header is not there
                    // Throw SOAP Fault exception
                    generateSoapFaultException(soapMsg);
                }
            } catch (SOAPException e) {
                e.printStackTrace();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        return true;
    }

    @Override
    public Set<QName> getHeaders() {
        // TODO Auto-generated method stub
        return null;
    }
```

```
        /**
         * This method is used to generate SOAP Fault Exception when secret key is
         * not matched
         * @param soapMsg
         */
        public void generateSoapFaultException(SOAPMessage soapMsg) {
                SOAPBody soapBody;
                try {
                        soapBody = soapMsg.getSOAPBody();
                        SOAPFault soapFault = soapBody.addFault();
                        soapFault.setFaultString("Secret Key is not matched...!!");
                        throw new SOAPFaultException(soapFault);
                } catch (SOAPException e) {
                        e.printStackTrace();
                }
        }
}
```

----------AuthenticateHandler.java End----------

Below is the Handler-Chain configuration file

```
<?xml version="1.0" encoding="UTF-8"?>
<handler-chains xmlns="http://java.sun.com/xml/ns/javaee">
      <handler-chain>
            <handler>
                  <handler-name>AuthenticateHandler</handler-name>
                  <handler-class>
                        com.atm.service.AuthenticateHandler
                  </handler-class>
            </handler>
      </handler-chain>
</handler-chains>
```

handler-chain.xml

Below are the Binding classes

Below is the Webservice description file

```
<?xml version="1.0" encoding="UTF-8"?>
<endpoints xmlns='http://java.sun.com/xml/ns/jax-ws/ri/runtime'
      version='2.0'>
      <endpoint name='IAtm'
            implementation='com.atm.service.AtmImpl'
            url-pattern='/withdraw' />
</endpoints>
```

sun-jaxws.xml

Web Application Deployment Descriptor file

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:web="http://xmlns.jcp.org/xml/ns/javaee">
  <listener>
    <listener-class>
      com.sun.xml.ws.transport.http.servlet.WSServletContextListener
    </listener-class>
  </listener>
  <servlet>
    <servlet-name>atm</servlet-name>
    <servlet-class>
            com.sun.xml.ws.transport.http.servlet.WSServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>atm</servlet-name>
    <url-pattern>/withdraw</url-pattern>
  </servlet-mapping>
</web-app>
```

web.xml

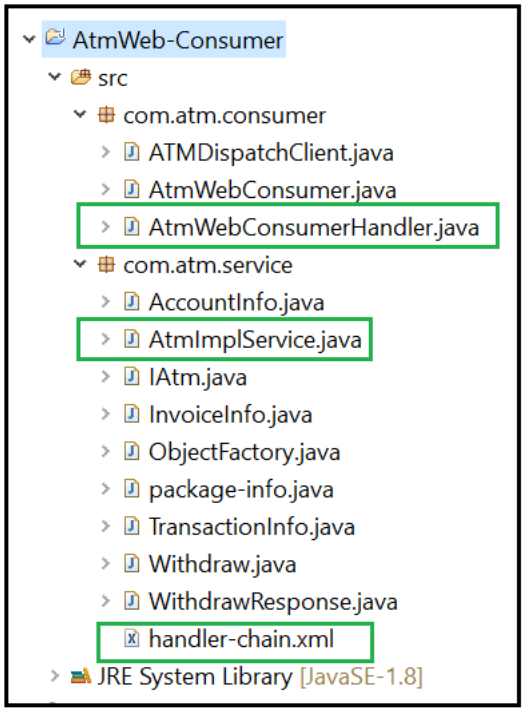
Deploy the Provider application and access WSDL with below URL

WSDL URL → http://localhost:4040/AtmWeb/withdraw?wsdl

If WSDL is accessible, provider is up and running

AtmWeb-Consumer Development with SOAP Handler

Create a project



Create Client-Side stubs using WSDL URL → **http://localhost:4040/AtmWeb/withdraw?wsdl**

Below is the Consumer Side Handler class which is used to add Secret Key into soap message

----------AtmWebConsumerHandler.java start----------

```
public class AtmWebConsumerHandler implements SOAPHandler<SOAPMessageContext> {
        public boolean handleMessage(SOAPMessageContext context) {
                System.out.println("Consumer : handleMessage() Begin");
                Boolean outBoundProperty =
                        (Boolean) context.get(MessageContext.MESSAGE_OUTBOUND_PROPERTY);
                // If its an outgoing message from client, then outBoundProperty will be
                // true
                if (outBoundProperty) {
                        try {
                                SOAPMessage soapMsg = context.getMessage();
                                SOAPEnvelope soapEnv = soapMsg.getSOAPPart().getEnvelope();
                                SOAPHeader soapHeader = soapEnv.getHeader();
```

```
                    // if no header, add the header
                    if (soapHeader == null) {
                        soapHeader = soapEnv.addHeader();
                    }
                    String secretKey = getSecretKey();
                    // add a soap header called "secretkey"
                    QName qname = new QName("http://service.atm.com/", "SecretKey");
                    SOAPHeaderElement soapHeaderElement =
                                soapHeader.addHeaderElement(qname);
                    soapHeaderElement.setActor(SOAPConstants.URI_SOAP_ACTOR_NEXT);
                    // Add Secret Key to SOAP header
                    soapHeaderElement.addTextNode(secretKey);
                    soapMsg.saveChanges();
                    // Output the message to console
                    soapMsg.writeTo(System.out);
                } catch (SOAPException e) {
                    System.err.println(e);
                } catch (IOException e) {
                    System.err.println(e);
                }
            }
            // Returning true makes other handler chain to continue the execution
            return true;
        }
        public boolean handleFault(SOAPMessageContext context) {
            System.out.println("Consumer : handleFault() Begin");
            return true;
        }
        public void close(MessageContext context) {
            System.out.println("Consumer : close() Begin");
        }
```

```
        public Set<QName> getHeaders() {
                System.out.println("Consumer : getHeaders() Begin");
                return null;
        }
        private String getSecretKey() {
                // This is the secret key which shared by provider
                return "Ashok-IT";
        }
}
```

----------AtmWebConsumerHandler.java End----------

```
<?xml version="1.0" encoding="UTF-8"?>
<handler-chains xmlns="http://java.sun.com/xml/ns/javaee">
      <handler-chain>
            <handler>
                  <handler-name>AtmWebConsumerHandler</handler-name>
                  <handler-class>
                        com.atm.consumer.AtmWebConsumerHandler
                  </handler-class>
            </handler>
      </handler-chain>
</handler-chains>
```

handler-chain.xml

Add the @HandlerChain annotation in AtmImplService.java class like below

```
@WebServiceClient(
      name = "AtmImplService",
      targetNamespace = "http://service.atm.com/",
      wsdlLocation = "http://localhost:4040/AtmWeb/withdraw?wsdl"
)
@HandlerChain(file="handler-chain.xml")
public class AtmImplService extends Service {

      //logic goes here
}
```

AtmImplService.java

Below is the Stub Based Consumer class to invoke Provider

JAX-WS API with APACHE-AXIS2 IMPLEMENTATION

Apache Axis2 is a Web Services / SOAP / WSDL engine, the successor to the widely used Apache Axis SOAP stack. There are two implementations of the Apache Axis2 Web services engine - Apache Axis2/Java and Apache Axis2/C.

Why Apache Axis2?

A new architecture for Axis2 was introduced during the August 2004 Summit in Colombo, Sri Lanka. The new architecture on which Axis2 is based on is more flexible, efficient and configurable in comparison to Axis1.x architecture. Some well-established concepts from Axis 1.x, like handlers etc., have been preserved in the new architecture.

Apache Axis2 not only supports SOAP 1.1 and SOAP 1.2, but it also has integrated support for the widely popular REST style of Web services. The same business logic implementation can offer both a WS-* style interface as well as a REST/POX style interface simultaneously.

Apache Axis2 is more efficient, more modular and more XML-oriented than the older version. It is carefully designed to support the easy addition of plug-in "modules" that extend their functionality for features such as security and reliability. The Modules currently available or under development include:

- WS-Security - Supported by Apache Rampart
- WS-Addressing -Module included as part of Axis2 core

Apache Axis2 is built on Apache AXIOM, a new high performant, pull-based XML object model.

Axis2 comes with many new features, enhancements and industry specification implementations. The key features offered are as follows:

Speed - Axis2 uses its own object model and StAX (Streaming API for XML) parsing to achieve significantly greater speed than earlier versions of Apache Axis.

Low memory foot print- Axis2 was designed ground-up keeping low memory foot print in mind.

AXIOM - Axis2 comes with its own light-weight object model, AXIOM, for message processing which is extensible, highly performant and is developer convenient.

Hot Deployment - Axis2 is equipped with the capability of deploying Web services and handlers while the system is up and running. In other words, new services can be added to the system without having to shut down the server. Simply drop the required Web service archive into the services directory in the repository, and the deployment model will automatically deploy the service and make it available for use.

Asynchronous Web services - Axis2 now supports asynchronous Web services and asynchronous Web services invocation using non-blocking clients and transports.

MEP Support - Axis2 now comes handy with the flexibility to support Message Exchange Patterns (MEPs) with in-built support for basic MEPs defined in WSDL 2.0.

Flexibility - The Axis2 architecture gives the developer complete freedom to insert extensions into the engine for custom header processing, system management, and anything else you can imagine.

Stability - Axis2 defines a set of published interfaces which change relatively slowly compared to the rest of Axis.

Component-oriented Deployment - You can easily define reusable networks of Handlers to implement common patterns of processing for your applications, or to distribute to partners.

Transport Framework - We have a clean and simple abstraction for integrating and using Transports (i.e., senders and listeners for SOAP over various protocols such as SMTP, FTP, message-oriented middleware, etc), and the core of the engine is completely transport-independent.

WSDL support - Axis2 supports the Web Service Description Language, version 1.1 and 2.0, which allows you to easily build stubs to access remote services, and also to automatically export machine-readable descriptions of your deployed services from Axis2.

Composition and Extensibility - Modules and phases improve support for composability and extensibility. Modules support composability and can also support new WS-* specifications in a simple and clean manner. They are however not hot deployable as they change the overall behavior of the system.

Apache Axis2 software is coming in the form of two distributions

1) Binary Distribution

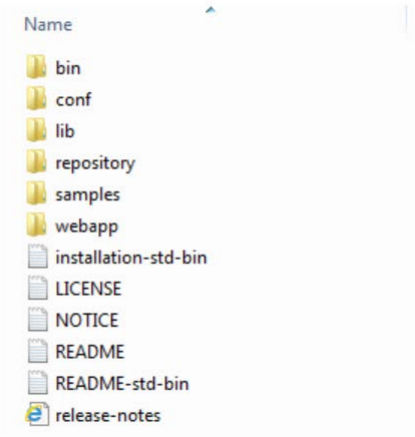
2) War Distribution

Binary Distribution: - which contains the Jar's and tools (Java2WSDL, WSDL2Java) that facilitate the development of Web Services.

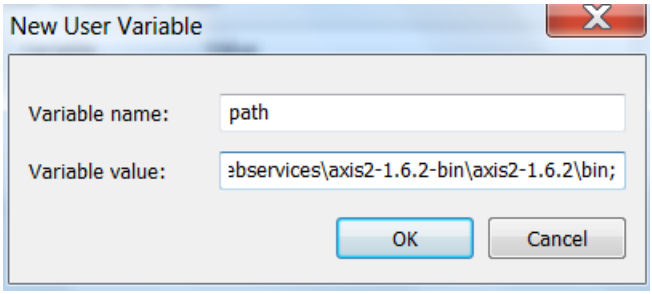
War Distribution: - Acts as Server-Side Runtime engine in which Your Services are deployed and exposed.

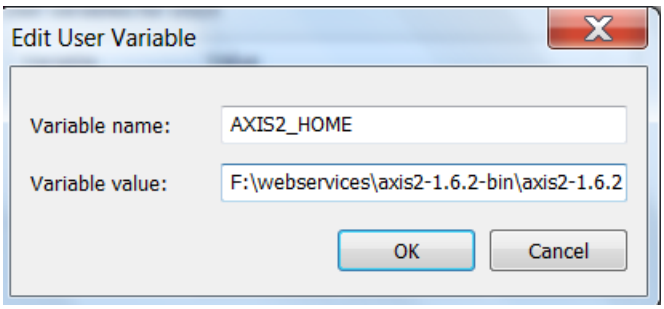
Download Apache AXIS2 Distribution from below URL

Extract it into some suitable location

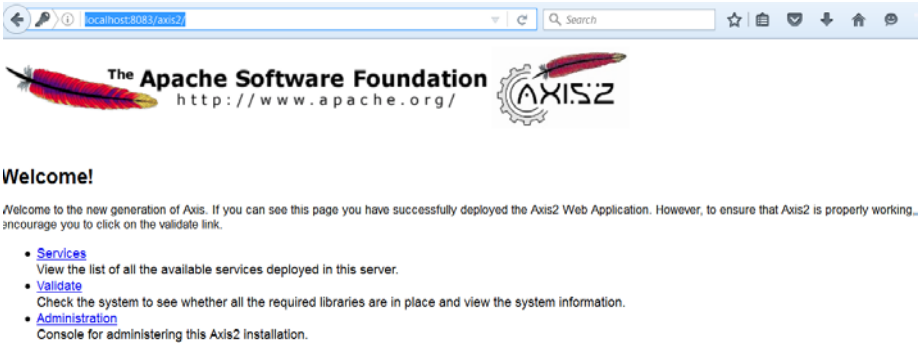


After Downloading the Axis2 Distributions first we can set path Environment variables of Axis2 binary distributions.



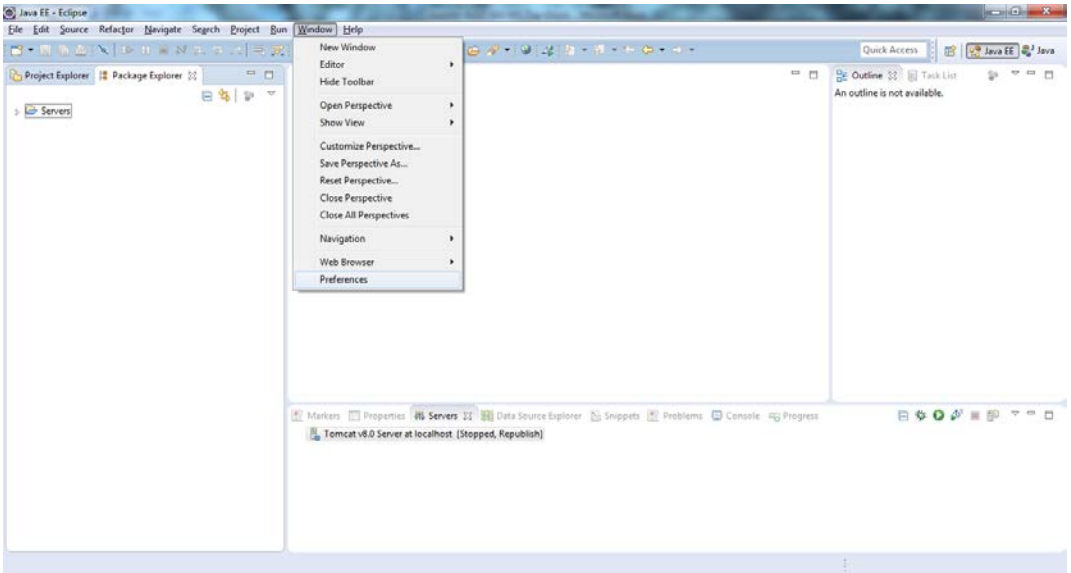


- After setting the Environment variables for binary distribution we can deploy the axis2.war file into any server
- Then Check axis2 welcome page is opening (OR) not.

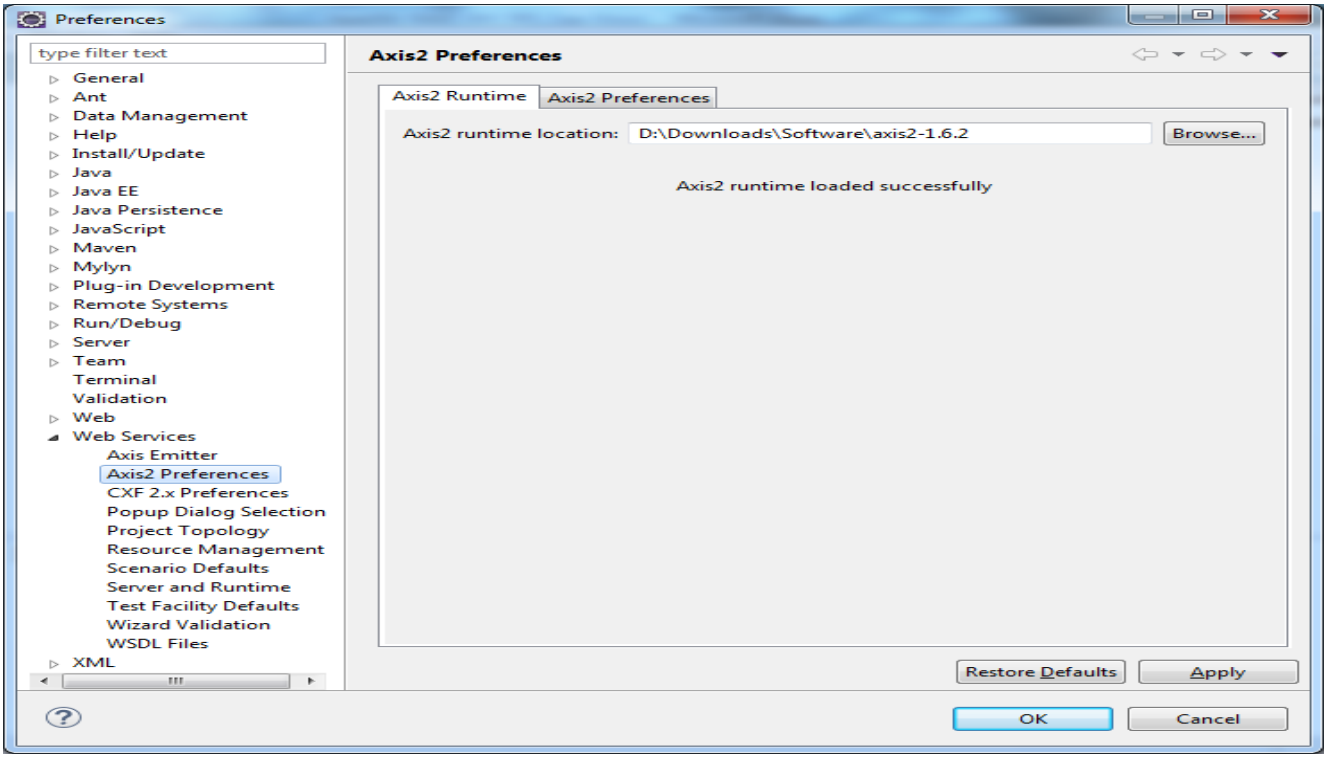


Configuring Apache AXIS-2 in Eclipse IDE

Step 1: Open Eclipse IDE Windows –> Preferences



Step 2: Browse through Axis-2 preferences and select axis-2 distribution - > Click on Apply - > Click on OK.



That's it!! we are done with installing Apache Axis2 plugin in Eclipse IDE

Now that we have installed Apache Axis2 plugin in Eclipse IDE and similarly configure Tomcat server we will move on developing top-down approach using above configuration.

Developing JAX-WS API WebService using Apache – Axis2 implementation

Technology Used

- Java 1.7
- Eclipse Kepler IDE
- Apache Axis2-1.6.2
- Apache Tomcat-8.0.12

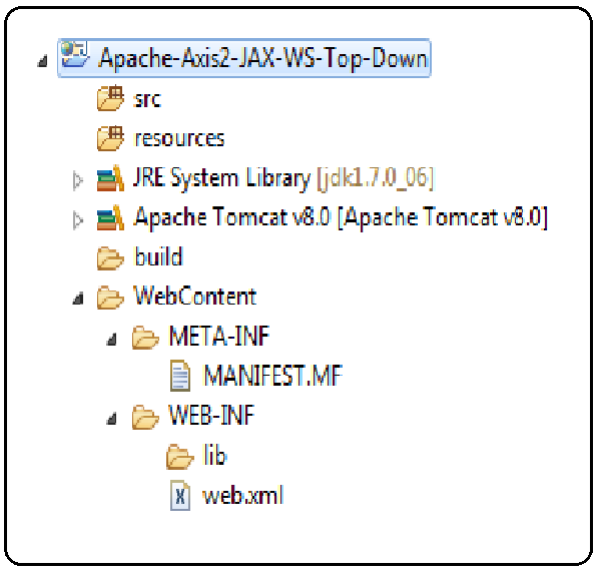
Step 1: In Eclipse, create new "Dynamic Web Project"

Step 2: Provide "Project Name" and make sure to change the "Dynamic web module version" to 2.5 –> click Next

Step 3: By default, there will be "src" folder –> add "resources" folder to place XSD/WSDL using "Add Folder…" button

Step 4: check "Generate web.xml" and Click Finish

Step 5: Initial project structure (Eclipse Package Explorer view)



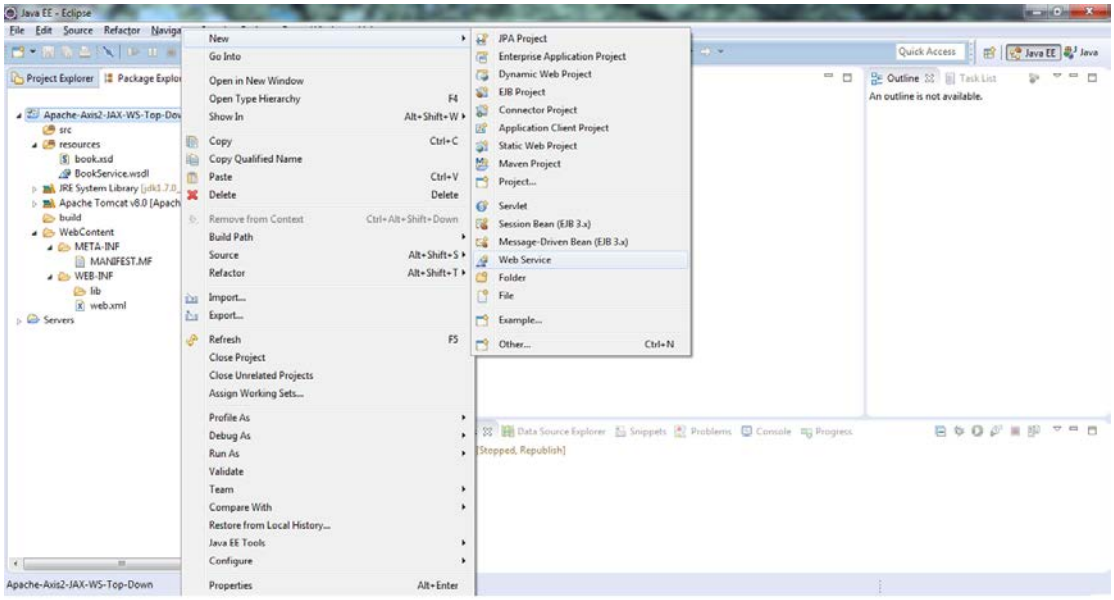
Step 6: Add Book Service WSDL and its associated XSD files under "resources" folder

book.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://amazon.com/sales/Book" xmlns:tns="http://amazon.com/sales/Book"
    elementFormDefault="qualified">

    <!-- Book Request Type -->
    <xsd:element name="BookRequestType">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="isbnNumber" type="xsd:string" />
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>

    <!-- Book Response Type -->
    <xsd:element name="BookResponseType">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="bookISBN" type="xsd:string" />
                <xsd:element name="bookName" type="xsd:string" />
                <xsd:element name="author" type="xsd:string" />
                <xsd:element name="category" type="xsd:string" />
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>

</xsd:schema>
```

BookService.wsdl

Step 7: Right Click on Project –> New –> Web Service



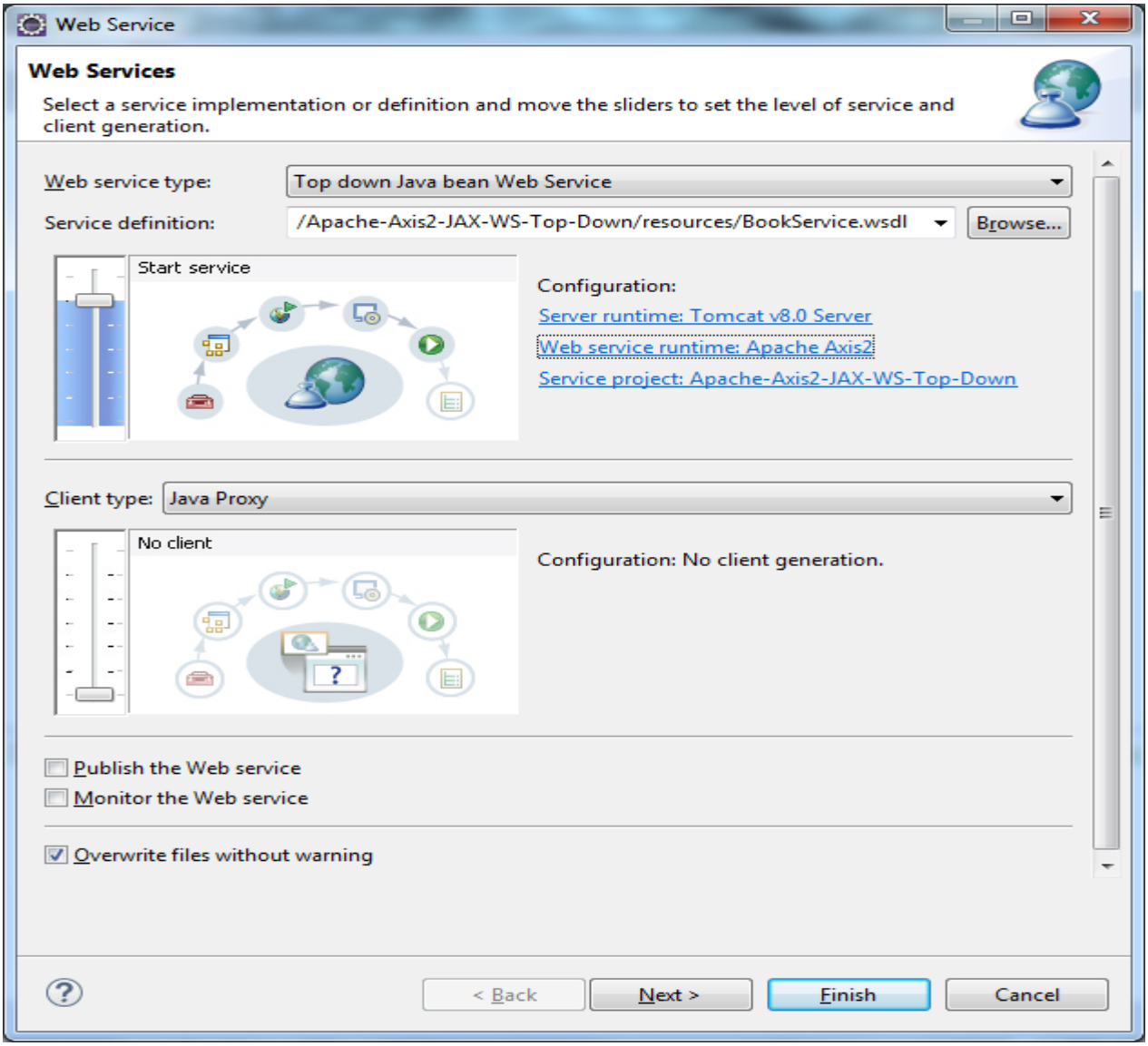
Step 9: Configure required parameters

Web service type: Top down Java bean Web Service

Service definition: browse through WSDL file here

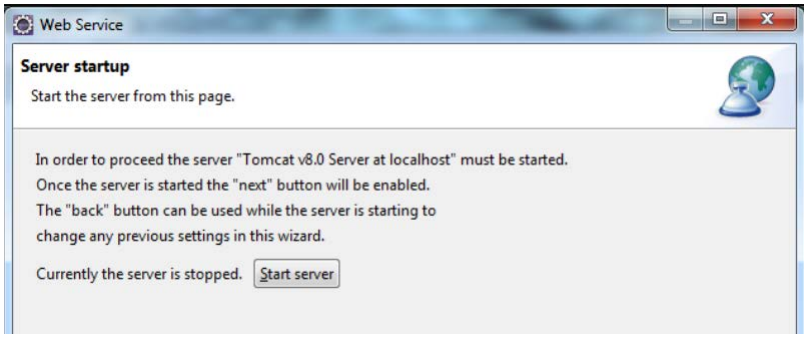
Configuration:

- ***Server runtime:*** Tomcat 8.x server
- ***Web service runtime:*** Apache Axis2
- ***Service project:*** Project-name



Step 11: accept default and click Next

Step 12: Click "*Start Server*"



Step 13: This is optional –> accept default and click Finish

Step 14: After clicking Finish in the above step -> default endpoint implementation class will be opened in Eclipse IDE

Step 15: Provide business logic to the endpoint class

Step 16: Web service implemented –> deploy the service to tomcat server

Deployment
Right click on project –> Run As –> Run on Server
Check whether web service is correctly deployed or not?

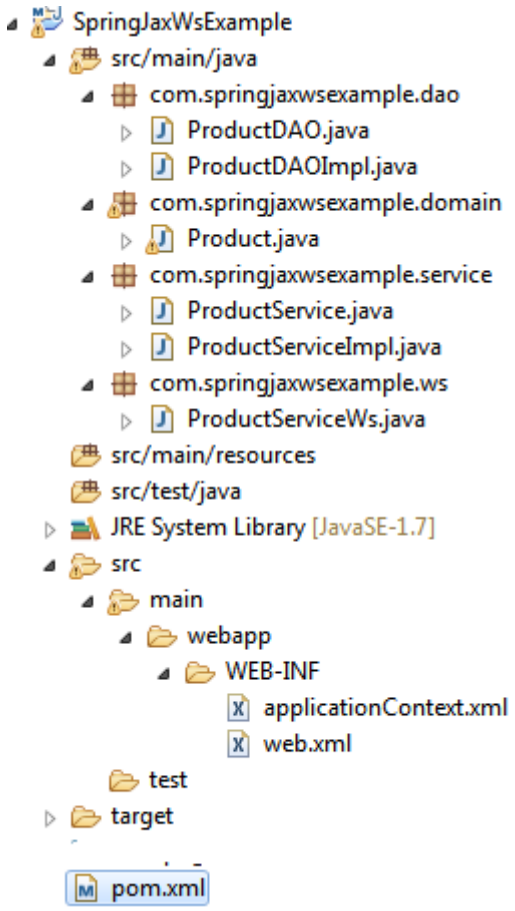
Step 17: This is home page, after deploying apache axis2 based web service in tomcat server. Click services

Step 18: Click BookService to view wsdl of the deployed book service

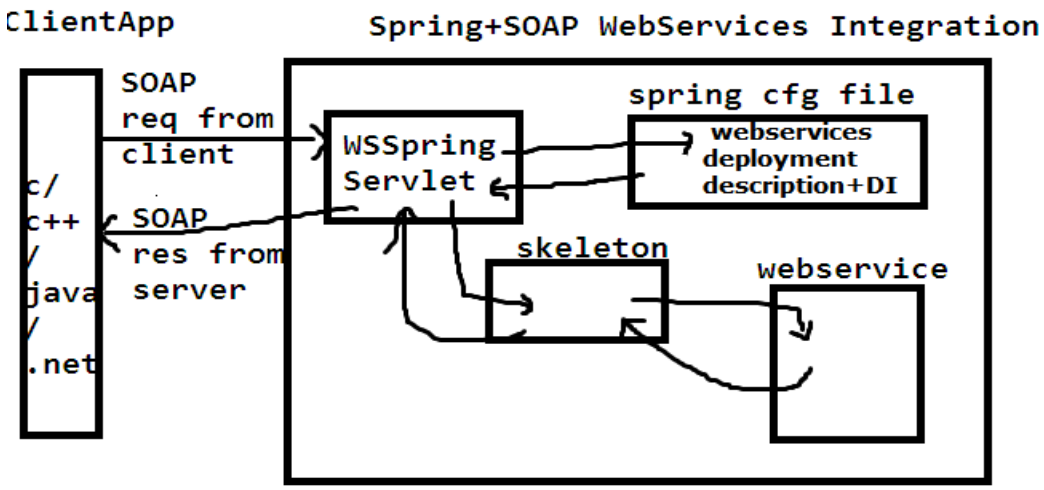
If we are able to see the WSDL in browser that means our provider is up and running.

We can test the provider using SOAP UI tool.

Spring +JAXWS Web Services (SOAP) Integration Example:




```
WsServlet,WsSpringServlet given by Jax-Ws-RI/Metro

WsServlet used to integrate JavaWebApp+WebServices app
WsSpringServlet used to integrate springapp+WebServices app
```



pom.xml

```
<projectxmlns="http://maven.apache.org/POM/4.0.0"xmlns:xsi="http://www.w3.org/2001/XML
Schema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-
v4_0_0.xsd">
<modelVersion>4.0.0</modelVersion>
<groupId>com.springjaxwsexample</groupId>
<artifactId>SpringJaxWsExample</artifactId>
<packaging>war</packaging>
<version>1.0</version>
<name>SpringJaxWsExample MavenWebapp</name>
<url>http://maven.apache.org</url>
<repositories>
<repository>
<id>java.net</id>
<url>http://download.java.net/maven/2</url>
</repository>
</repositories>
<dependencies>
<dependency>
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>3.8.1</version>
<scope>test</scope>
</dependency>
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-web</artifactId>
<version>3.2.0.RELEASE</version>
</dependency>
<!-- JAX-WS-RI -->
        <dependency>
                <groupId>com.sun.xml.ws</groupId>
        <artifactId>jaxws-rt</artifactId>
        <version>2.2.3</version>
        </dependency>
        <!-- Library from java.net, integrate Spring with JAX-WS -->
        <dependency>
                <groupId>org.jvnet.jax-ws-commons.spring</groupId>
                <artifactId>jaxws-spring</artifactId>
                <version>1.8</version>
                <exclusions>
                <exclusion>
                <groupId>org.springframework</groupId>
                        <artifactId>spring-core</artifactId>
        </exclusion>
        <exclusion>
```

```
<groupId>org.springframework</groupId>
                        <artifactId>spring-context</artifactId>
        </exclusion>
        <exclusion>
                <groupId>com.sun.xml.stream.buffer</groupId>
                        <artifactId>streambuffer</artifactId>
        </exclusion>
        <exclusion>
                <groupId>org.jvnet.staxex</groupId>
                        <artifactId>stax-ex</artifactId>
        </exclusion>
                <groupId>org.jvnet.staxex</groupId>
                        <artifactId>stax-ex</artifactId>
        </exclusion>
                </exclusions>
        </dependency>
        <dependency>
<groupId>mysql</groupId>
<artifactId>mysql-connector-java</artifactId>
<version>5.1.35</version>
</dependency>
<!-- https://mvnrepository.com/artifact/commons-pool/commons-pool -->
<dependency>
<groupId>commons-pool</groupId>
<artifactId>commons-pool</artifactId>
<version>1.6</version>
</dependency>
<!-- https://mvnrepository.com/artifact/commons-dbcp/commons-dbcp -->
<dependency>
<groupId>commons-dbcp</groupId>
<artifactId>commons-dbcp</artifactId>
<version>1.4</version>
</dependency>
</dependencies>
<build>
<finalName>SpringJaxWsExample</finalName>
</build>
</project>
```

web.xml

```
<web-app>
<servlet>
        <servlet-name>jaxwsservlet</servlet-name>
        <servlet-class>
 com.sun.xml.ws.transport.http.servlet.WSSpringServlet
        </servlet-class>
        <load-on-startup>1</load-on-startup>
        </servlet>
        <servlet-mapping>
<servlet-name>jaxwsservlet</servlet-name>
<url-pattern>/registerProduct</url-pattern>
```

```
</servlet-mapping>
<!-- Register Spring Listener -->
        <listener>
        <listener-class>
                org.springframework.web.context.ContextLoaderListener
        </listener-class>
        </listener>
</web-app>
```

Note : ContextLoaderListener will create spring container.It searches the spring cfg file name as applicationContext.xml in WEB-INF folder

Product.java

```
package com.springjaxwsexample.domain;
import java.io.Serializable;
publicclassProductimplements Serializable{
private Integer pid;
private String productName;
private Double price;
public Integer getPid() {
        returnpid;
}
publicvoid setPid(Integer pid) {
        this.pid = pid;
}
public String getProductName() {
        returnproductName;
}
publicvoid setProductName(String productName) {
        this.productName = productName;
}
public Double getPrice() {
        returnprice;
}
publicvoid setPrice(Double price) {
        this.price = price;
}
}
```

ProductDAO.java

```
package com.springjaxwsexample.dao;
import com.springjaxwsexample.domain.Product;
publicinterface ProductDAO {
publicint registerProduct(Product product);
}
```

ProductDAOImpl.java

```
package com.springjaxwsexample.dao;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import javax.sql.DataSource;
import com.springjaxwsexample.domain.Product;
public class ProductDAOImpl implements ProductDAO{
private DataSource dataSource;//dependency
	public void setDataSource(DataSource dataSource) {
	this.dataSource = dataSource;
}
	publicint registerProduct(Product product) {
	intcount=0;
		try{
	Connection con=dataSource.getConnection();
String sql="insert into Product_Details values(?,?,?)";
PreparedStatement pst=con.prepareStatement(sql);
pst.setInt(1,product.getPid());
pst.setString(2,product.getProductName());
pst.setDouble(3,product.getPrice());
count=pst.executeUpdate();
		}catch(SQLException se){
		se.printStackTrace();
		}
		returncount;
	}
	}
```

ProductService.java

```
package com.springjaxwsexample.service;
import com.springjaxwsexample.domain.Product;
public interface ProductService {
public boolean registerProduct(Product product);
}
```

ProductServiceImpl.java

```
package com.springjaxwsexample.service;
import com.springjaxwsexample.dao.ProductDAO;
import com.springjaxwsexample.domain.Product;
/** writing of this service is an optional
 * in this class write Exception-Handling,Transaction-Management logic,loggers,Type-Converstion,Bussiness logic's
 * if we are not writing this separate service class then we include above logic directly in webservice class
```

```
*/
publicclass ProductServiceImpl implements  ProductService{
private ProductDAO productDAO;
public void setProductDAO(ProductDAO productDAO) {
        this.productDAO = productDAO;
}
public cboolean registerProduct(Product product) {
        booleanflag=false;
        intcount=productDAO.registerProduct(product);
        if(count>0){
                flag=true;
         }
                returnflag;
        }
```

ProductServiceWS.java

```
package com.springjaxwsexample.ws;
import javax.jws.WebMethod;
import javax.jws.WebService;
import com.springjaxwsexample.domain.Product;
import com.springjaxwsexample.service.ProductService;
@WebService
publicclass ProductServiceWs{
        private ProductService productService;
        @WebMethod(exclude=true)
public void setProductService(ProductService productService) {
        this.productService = productService;
}
@WebMethod(exclude=false)
        public Boolean registerProduct(Product product) {
                Boolean flag=productService.registerProduct(product);
                returnflag;
        }
}
```

applicationContext.xml

```
<beansxmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:ws="http://jax-ws.dev.java.net/spring/core"
xmlns:wss="http://jax-ws.dev.java.net/spring/servlet"
xsi:schemaLocation="http://www.springframework.org/schema/beans
     http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
     http://jax-ws.dev.java.net/spring/core
      http://jax-ws.dev.java.net/spring/core.xsd
      http://jax-ws.dev.java.net/spring/servlet
      http://jax-ws.dev.java.net/spring/servlet.xsd">
```

```
<wss:bindingurl="/registerProduct">
<wss:service>
<ws:servicebean="#productServiceWs"/>
</wss:service>
</wss:binding>
<beanid="dataSource"class="org.apache.commons.dbcp.BasicDataSource">
<propertyname="driverClassName"value="com.mysql.jdbc.Driver"/>
<propertyname="url"value="jdbc:mysql://localhost:3306/nit"/>
<propertyname="username"value="root"/>
<propertyname="password"value="root"/>
</bean>
<beanid="productDAO"class="com.springjaxwsexample.dao.ProductDAOImpl">
<propertyname="dataSource"ref="dataSource"/>
</bean>
<beanid="productService"class="com.springjaxwsexample.service.ProductServiceImpl">
<propertyname="productDAO"ref='productDAO'/>
</bean>
<beanid="productServiceWs"class="com.springjaxwsexample.ws.ProductServiceWs">
<propertyname="productService"ref="productService"/>
</bean>
</beans>
```

Note: Test the above service by using SOAP UI tool.

Restful Services

The term REST comes from Roy Fielding's PhD dissertation, published in 2000, and it stands for **REpresentational State Transfer.** REST by itself is not an architecture; REST is a set of constraints that, when applied to the design of a system, creates a software architectural style. If we implement all the REST guidelines outlined in Fielding's work, we end up with a system that has specific roles for data, components, hyperlinks, communication protocols, and data consumers.

Fielding arrived at REST by evaluating all networking resources and technologies available for creating distributed applications. Without any constraints, anything and everything goes, leading us to develop applications that are hard to maintain and extend. He looks at document distributed application architectural styles beginning with what he calls the null space—which represents the availability of every technology and every style of application development with no rules or limits—and ends with the following constraints that define a RESTful system:

- ➔ It must be a client-server system
- ➔ It has to be stateless—there should be no need for the service to keep users' sessions; in other words, each request should be independent of others
- ➔ It has to support a caching system—the network infrastructure should support cache at different levels
- ➔ It has to be uniformly accessible—each resource must have a unique address and a valid point of access
- ➔ It has to be layered—it must support scalability
- ➔ It should provide code on demand—although this is an optional constraint, applications can be extendable at runtime by allowing the downloading of code on demand, for example, Java Applets

These constraints don't dictate what kind of technology to use; they only define how data is transferred between components and what are the benefits of following the guidelines. Therefore, a RESTful system can be implemented in any networking architecture available. More important, there is no need for us to invent new technologies or networking protocols: we can use existing networking infrastructures such as the Web to create RESTful architectures. Consequently, a RESTful architecture is one that is maintainable, extendable, and distributed.

Before all REST constraints were formalized, we already had a working example of a RESTful system: that ie Web. We can ask, then, why to introduce these RESTful requirements to web application development when it's agreed that the Web is already RESTful ? - We need to first qualify here what it's meant for the Web to be RESTful. On the one hand, the static web is RESTful, because static websites follow Fielding's definition of a RESTful architecture. For instance, the existing web infrastructure provides caching systems, stateless connection, and unique hyperlinks to resources, where resources are all of the documents available on every website and the representations of these documents are already set by files being browser readable (HTML files, for example).

Therefore, the static web is a system built on the REST-like architectural style. On the other hand, traditional dynamic web applications haven't always been RESTful, because they typically break some of

the outlined constraints. For instance, most dynamic applications are not stateless, as servers require tracking users through container sessions or client-side cookie schemes. Therefore, we conclude that the dynamic web is not normally built on the REST-like architectural style. We can now look at the

abstractions that make a RESTful system, namely resources, representations, URIs, and the HTTP request types that make up the uniform interface used for client/server data transfers.

Resources

A RESTful resource is anything that is addressable over the Web. By addressable, we mean resources that can be accessed and transferred between clients and servers. Subsequently, a resource is a logical, temporal mapping to a concept in the problem domain for which we are implementing a solution.

These are some examples of REST resources:

- A news story
- The temperature in NY at 4:00 p.m. EST
- A tax return stored in IRS databases
- A list of code revisions history in a repository like SVN or CVS
- A student in some classroom in some school
- A search result for a item in a web index, such as Google

Even though a resource's mapping is unique, different requests for a resource can return the same underlying binary representation stored in the server. For example, let's say we have a resource within the context of a publishing system. Then, a request for "the latest revision published" and the request for "revision number 12" will at some point in time return the same representation of the resource: the last revision is revision 12. However, when the latest revision published is increased to version 13, a request to the latest revision will return version 13, and a request for revision 12 will continue returning version 12. As resources in a RESTful architecture, each of these resources can be accessed directly and independently, but different requests could point to the same data.

Because we are using HTTP to communicate, we can transfer any kind of information that can be passed between clients and servers. For example, if we request a text file from CNN, our browser receives a text file. If we request a Flash movie from YouTube, our browser receives a Flash movie. The data is streamed in both cases over TCP/IP and the browser knows how to interpret the binary streams because of the HTTP protocol response header Content-Type. Consequently, in a RESTful system, the representation of a resource depends on the caller's desired type (MIME type), which is specified within the communication protocol's request.

Representation

The representation of resources is what is sent back and forth between clients and servers. A representation is a temporal state of the actual data located in some storage device at the time of a request. In general terms, it's a binary stream together with its metadata that describes how the stream is to be consumed by either the client or the server (metadata can also contain extra information about the resource, for example, validation, encryption information, or extra code to be executed at runtime).

Throughout the life of a web service there may be a variety of clients requesting resources. Different clients are able to consume different representations of the same resource. Therefore, a representation can take various forms, such as an image, a text file, or an XML stream or a JSON stream, but has to be available through the same URI.

For human-generated requests through a web browser, a representation is typically in the form of an HTML page. For automated requests from other web services, readability is not as important and a more efficient representation can be used such as XML.

URI

A Uniform Resource Identifier, or URI, in a RESTful web service is a hyperlink to a resource, and it's the only means for clients and servers to exchange representations. The set of RESTful constraints don't dictate that URIs must be hyperlinks. We only talk about RESTful URIs being hyperlinks, because we are using the Web to create web services. If we were using a different set of supporting technologies, a RESTful URI would look completely different. However, the core idea of addressability would still remain.

In a RESTful system, the URI is not meant to change over time, as the architecture's implementation is what manages the services, locates the resources, negotiates the representations, and then sends back responses with the requested resources. More important, if we were to change the structure of the storage device at the server level (swapping database servers, for example), our URIs will remain the same and be valid for as long the web service is online or the context of a resource is not changed.

Without REST constraints, resources are accessed by location: typical web addresses are fixed URIs. For instance, if we rename a file on a web server, the URI will be different; if we move a file to a different directory tree in a web server, the URI will change. Note that we could modify our web servers to execute redirects at runtime to maintain addressability, but if we were to do this for every file change, our rules would become unmanageable.

Uniform interfaces through HTTP requests

In previous sections, we introduced the concepts of resources and representations. We said that resources are mappings of actual entity states that are exchanged between clients and servers. Furthermore, we discussed that representations are negotiated between clients and servers through the communication protocol at runtime—through HTTP. In this section, we look in detail at what it means to exchange these representations, and what it means for clients and servers to take actions on these resources.

Developing RESTful web services is similar to what we've been doing up to this point with our web applications. However, the fundamental difference between modern and traditional web application development is how we think of the actions taken on our data abstractions. Specifically, modern development is rooted in the concept of nouns (exchange of resources); legacy development is rooted in the concept of verbs (remote actions taken on data). With the former, we are implementing a RESTful web service; with the latter, we are implementing an RPC-like service (Remote Procedure Call). What's more, a RESTful service modifies the state of the data through the representation of resources; an RPC service, on the other hand, hides the data representation and instead sends commands to modify the state of the data at the server level (we never know what the data looks like).

Finally, in modern web application development we limit design and implementation ambiguity, because we have four specific actions that we can take upon resources—Create, Retrieve, Update, and Delete (CRUD). On the other hand, in traditional web application development, we can have countless actions with no naming or implementation standards.

Therefore, with the delineated roles for resources and representations, we can now map our CRUD actions to the HTTP methods POST, GET, PUT, and DELETE as follows:

Data action	HTTP protocol equivalent
CREATE	POST
RETRIEVE	GET
UPDATE	PUT
DELETE	DELETE

In their simplest form, RESTful web services are networked applications that manipulate the state of resources. In this context, resource manipulation means resource creation, retrieval, update, and deletion. However, RESTful web services are not limited to just these four basic data manipulation concepts. On the contrary, RESTful web services can execute logic at the server level, but remembering that every result must be a resource representation of the domain at hand.

Let's now look at the four HTTP request types in detail and see how each of them is used to exchange representations to modify the state of resources.

GET/RETRIEVE: The method GET is used to RETRIEVE resources.

Before digging into the actual mechanics of the HTTP GET request, first, we need to determine what a resource is in the context of our web service and what type of representation we're exchanging.

For the rest of this section, we'll use the artificial example of a web service handling students in some classroom, with a location of http://ashokit.com/. For this service, we assume an XML representation of a student to look as follows:

```
<student>
        <name>Jane</name>
        <age>10</age>
        <link>/students/Jane</link>
</student>
```

And a list of students to look like:

```
<students>
        <student>
                <name>Jane</name>
                <age>10</age>
                <link>/students/Jane</link>
        </student>
        <student>
                <name>John</name>
                <age>11</age>
                <link>/students/John</link>
                </student>
        <link>/students</link>
</students>
```

With our representations defined, we now assume URIs of the form **http://ashokit.com/students** to access a list of students, and **http://ashokit.com/students/{name}** to access a specific student that has the unique identifier of value name.

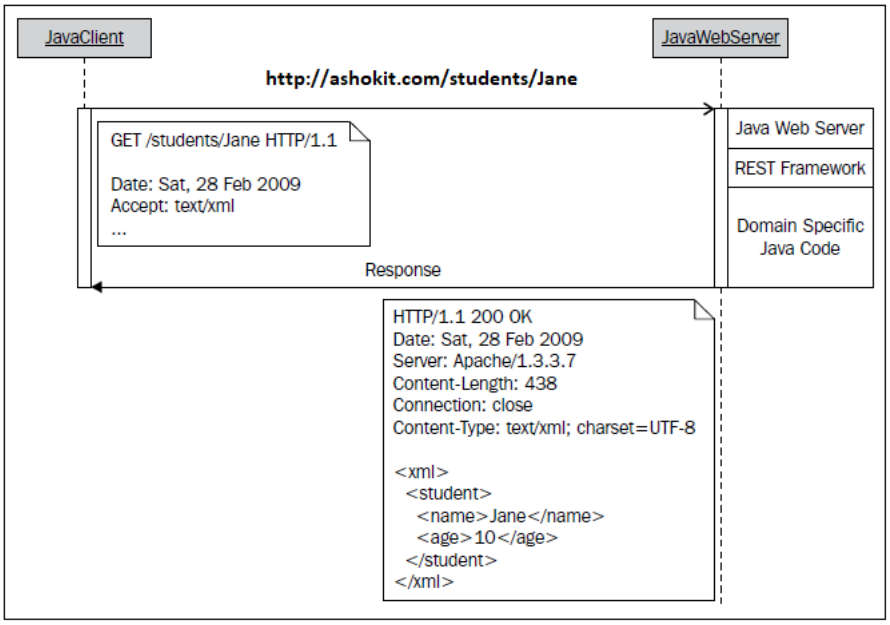
We can now begin making requests to our web service. For instance, if we wanted the record for a student with the name Jane, we make a request to the URI **http://ashokit.com/students/Jane**. A representation of Jane, at the time of the request, may look like:

```
<student>
        <name>Jane</name>
        <age>10</age>
        <link>/students/Jane</link>
</student>
```

Subsequently, we can access a list of students through the URI **http://ashokit.com/students**. The response from the service will contain the representation of all students and may look like (assuming there are two students available):

```
<students>
        <student>
                <name>Jane</name>
                <age>10</age>
                <link>/students/Jane</link>
        </student>
        <student>
                <name>John</name>
                <age>11</age>
                <link>/students/John</link>
                </student>
        <link>/students</link>
</students>
```

Now let's have a look at the request details. A request to retrieve a Jane resource uses the GET method with the URI **http://ashokit.com/students/Jane**. A sequence diagram of our GET request looks as follows:



What's happening in the above GET request?

1. A Java client makes an HTTP request with the method type GET and Jane as the identifier for the student.
2. The client sets the representation type it can handle through the Accept request header field.
3. The web server receives and interprets the GET request to be a retrieve action. At this point, the web server passes control to the RESTful framework to handle the request. Note that RESTful frameworks don't automatically retrieve resources, as that's not their job. The job of a framework is to ease the implementation of the REST constraints. Business logic and storage implementation is the role of the domain-specific Java code.
4. The server-side program looks for the Jane resource. Finding the resource could mean looking for it in a database, a filesystem, or a call to a different web service.
5. Once the program finds Jane, it converts the binary data of the resource to the client's requested representation.
6. With the representation converted to XML, the server sends back an HTTP response with a numeric code of 200 together with the XML representation as the payload. Note that if there are any errors, the HTTP server reports back the proper numeric code, but it's up to the client to correctly deal with the failure.

All the messages between client and server are standard HTTP protocol calls. For every retrieve action, we send a GET request and we get an HTTP response back with the payload of the response being the representation of the resource or, if there is a failure, a corresponding HTTP error code (for example, 404 if a resource is not found; 500, if there is a problem with the Java code in the form of an exception).

The HTTP GET method should only be used to retrieve representations. As we know, we can use a GET request to update the state of data in the server, but this is not recommended. A GET request must be safe and idempotent. (For more information, see http://www.w3.org/DesignIssues/Axioms.)

For a request to be safe, it means that multiple requests to the same resource don't change the state of the data in the server. Assume we have a representation R and requests happen at time t. Then, a request at time t1 for resource R returns R1; subsequently, a request at time t2 for resource R returns R2; provided that no further update actions have been taken between t1 and t2, then R1 = R2 = R.

For a request to be idempotent, it means that multiple calls to the same action don't change the state of the resource. For example, multiple calls to create a resource R at time t1, t2, and t3 means that R will exist only as R and that calls at times t2 and t3 are ignored.

POST/CREATE: The method POST is used to CREATE resources.

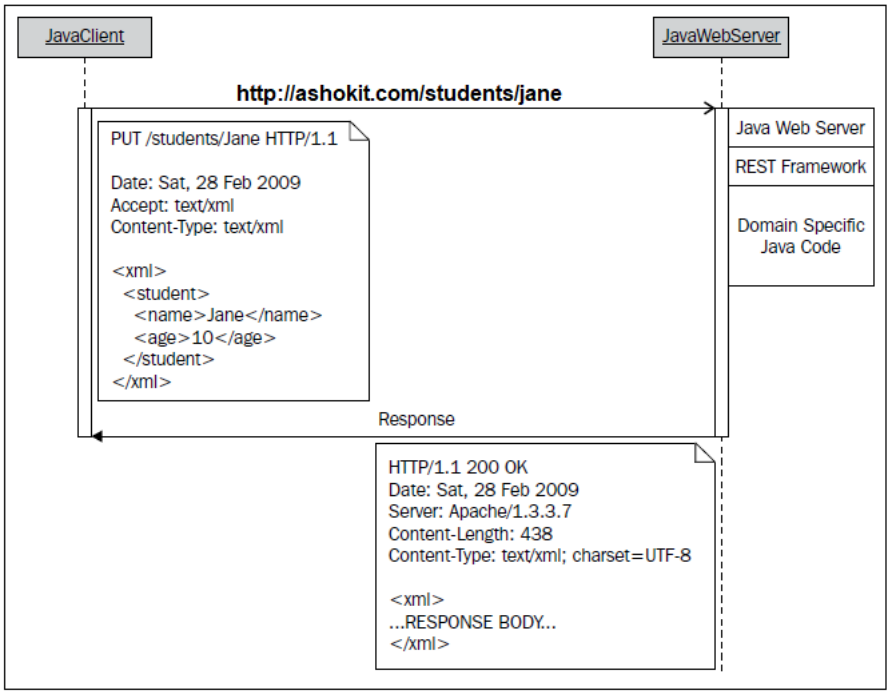
Because we are creating a student, we use the HTTP method POST. Again, the URI to create a new student in our list is http://restfuljava.com/students/Jane. The method type for the request is set by the client.

Assume Jane didn't exist in our list, and we want to add her. Our new Jane XML representation looks like:

```
<student>
        <name>Jane</name>
        <age>10</age>
        <link></link>
</student>
```

Note that the link element is part of the representation, but it's empty because this value is generated at runtime and not created by the client sending the POST request. This is just a convention for this example; however, clients using a web service can dictate the structure of URIs.

Now, the sequence diagram of our POST request looks as follows:



What's happening in above POST request?

1. A Java client makes a request to the URI http://ashokit.com/students/Jane, with the HTTP method set to POST.

2. The POST request carries with it the payload in the form of an XML representation of Jane.

3. The server receives the request and lets the REST framework handle it; our code within the framework executes the proper commands to store the representation (again, the storage device could be anything).

4. Once storing of the new resource is completed, a response is sent back: if it's successful, we send a code of 200; if it's a failure, we send the appropriate error code.

PUT/UPDATE: The method PUT is used to UPDATE resources.

To update a resource, we first need its representation in the client; second, at the client level we update the resource with the new value(s) we want; and, finally, we update the resource using a PUT request together with the representation as its payload.

We're omitting the GET request to retrieve Jane from the web service, as it's the same one we illustrated in the previous section. We must, however, modify the representation at the client level first. Assume that we already have the student representation of Jane in the client and we want to change her age from 10 to 12.

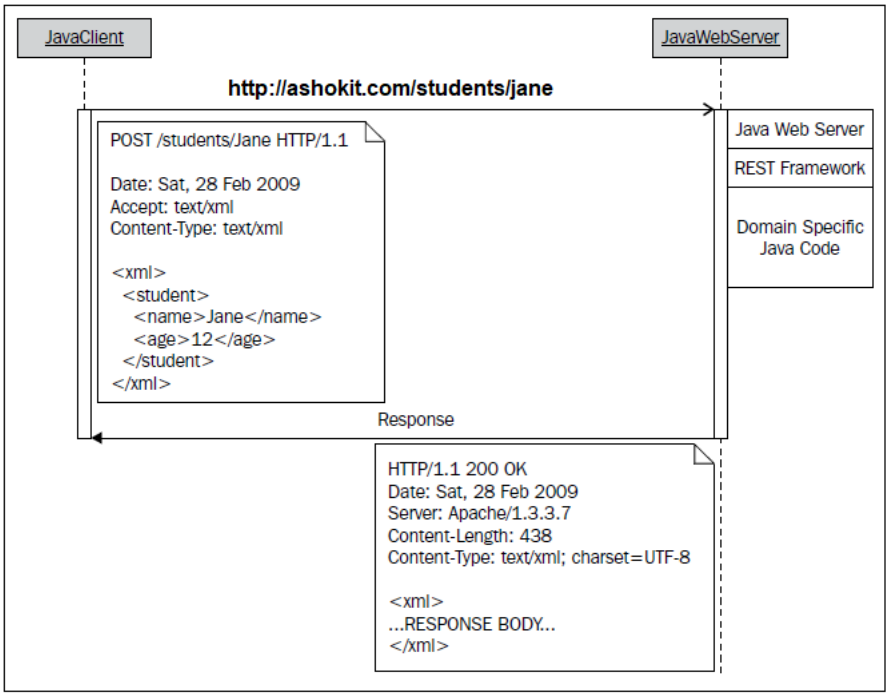
Our original student representation looks as follows:

```
<student>
        <name>Jane</name>
        <age>10</age>
        <link>/students/Jane</link>
</student>
```

Changing Jane's age to 12, our representation looks as follows:

```
<student>
        <name>Jane</name>
        <age>12</age>
        <link>/students/Jane</link>
</student>
```

We are now ready to connect to our web service to update Jane by sending the PUT request to http://ashokit.com/students/Jane. The sequence diagram of our PUT request looks as follows:



What's happening in above PUT request?

1. A Java client makes a PUT request to http://ashokit.com/students/Jane, with the new XML definition as its payload.

2. The server receives the request and lets the REST framework handle it. At this point, we let our code execute the proper commands to update the representation of Jane.

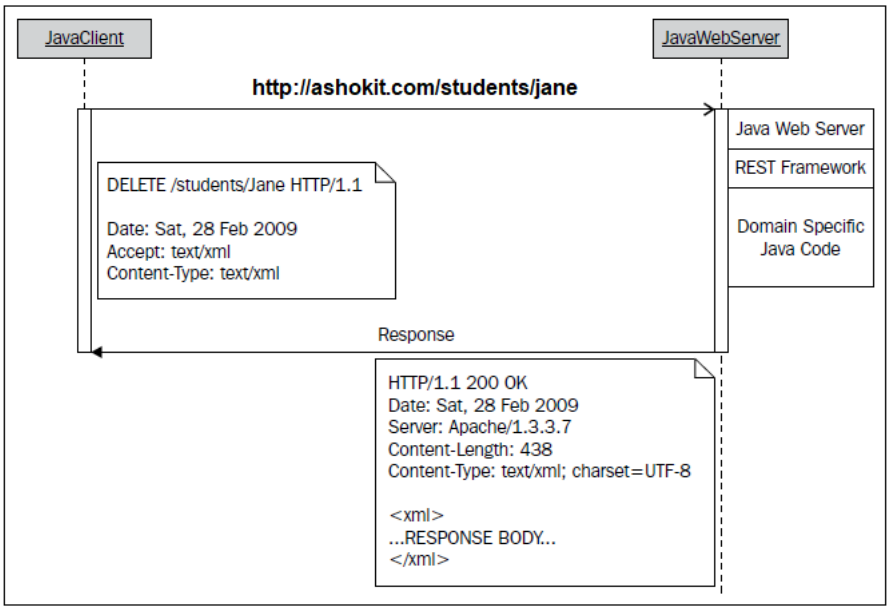
3. Once completed, a response is sent back.

DELETE/DELETE: The method DELETE is used to DELETE representations.

Finally, deleting a resource makes use of the same URI we've used in the other three cases.

Assume we want to delete Jane from our data storage. We send a DELETE request to our service with the URI **http://ashokit.com/students/Jane**.

The sequence diagram for our DELETE request looks like:



What's happening in above DELETE request?

1. A Java client makes a DELETE request to http://ashokit.com/students/Jane.

2. The server receives the request and lets the REST framework handle it. At this point, our code executes the proper commands to delete the representation of Jane.

3. Once completed, a response is sent back.

JAX-RS-> JAX-RS is API from sun .

- JAX-RS API is used to develop/access the Restful services in java platform.
- **JAX-RS** stands for Java API For Xml Restfull services.
- JAX-RS Versions 0.x ,1.x (1.0, 1.1), 2.x
- JAX-RS 2.0 is the latest release of JAX-RS.
- JAX-RS uses annotations, introduced in Java SE 5, to simplify the development and deployment of Restful services resources and clients.
- From version JAX-RS 1.1 on words , JAX-RS is an official part of Java EE 6
- The Java API for RESTful Services provides portable APIs for developing, exposing and accessing Web applications designed and implemented in compliance with principles of REST architectural style.

Packages

Package	Description
javax.ws.rs	High-level interfaces and annotations used to create RESTful service resources.
javax.ws.rs.client	The JAX-RS client API
javax.ws.rs.container	Container-specific JAX-RS API.
javax.ws.rs.core	Low-level interfaces and annotations used to create RESTful service resources.
javax.ws.rs.ext	APIs that provide extensions to the types supported by the JAX-RS API.

Implementations of Jax RS API:-

- Apache CXF, an open source Web service framework
- Jersey, the reference implementation from Sun (now Oracle)
- RESTeasy, JBoss's implementation
- Restlet from jerome Louvel.
- Apache Wink, Apache Software Foundation

Difference between SOAP and Restful services

- In soap webservices we can use only xml to make communication between the client and server provider.
- In Restfull webservice we can make communitaction in between server provider and client using
    - XML
    - JSON (javascriptObjectNotation)
    - text
    - HTML
- Soap WebService supports both server level and application level security
- But Restful services support only server level security but not for the Application level security.

What is a Resource?

In REST architecture, everything is a resource. These resources can be text files, html pages, images, videos or dynamic business data. REST Server simply provides access to resources and REST client accesses and modifies the resources. Here each resource is identified by URIs. REST uses various

representations to represent a resource like text, JSON, XML. XML and JSON are the most popular representations of resources.

Representation of Resources

A resource in REST is similar Object in Object Oriented Programming. Once a resource is identified then its representation is to be decided using a standard format so that server can send the resource in above said format and client can understand the same format.

For example, in RESTful Web Services - User is a resource which is represented using **following XML format:**

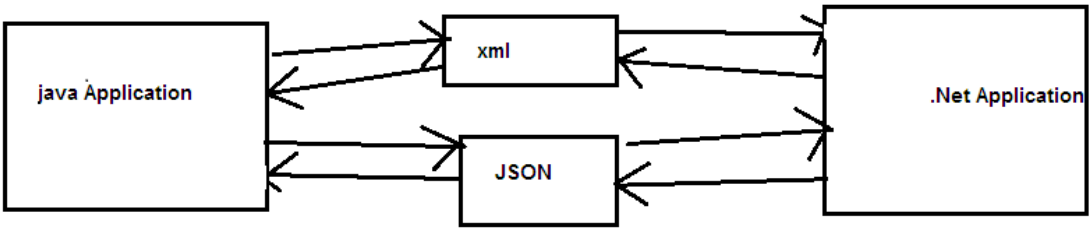
```
<user>
  <id>101</id>
  <name>Mahesh</name>
  <profession>Former</profession>
</user>
```

Same resource can be represented in JSON format:

```
{
  "id":101,
  "name":"Mahesh",
  "profession":"Former"
}
```

JSON:

- JSON stands for JavaScript Object Notation
- JSON is also communicator like xml is between two language Applications.



- we can apply Restriction xml document to accept the data by using DTD or XSD.
- JSON we can't restrict to accept the data because it is normal text document.
- JSON format is interoperable format
- interoperable->lang independent and platform independent
- JSONformat is very lightweight format compared to SOAP/XML.
- JSON a is text-based document

Ex:-

{"studentId":101,"name":"raja","course":"java"}

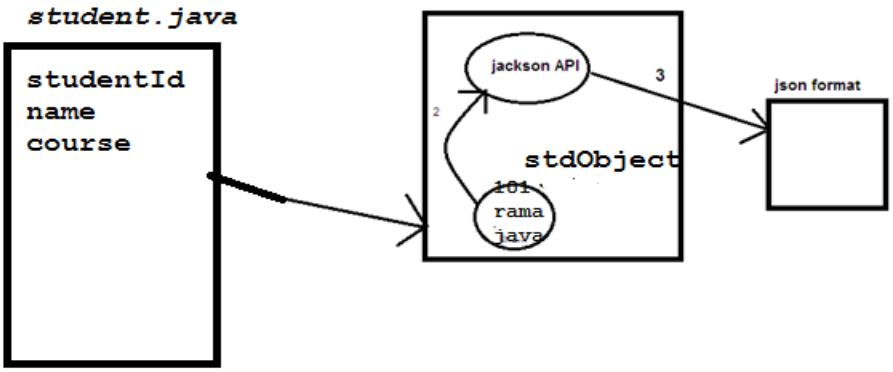
➔ To read the JSON format data from the java application and to construct JSON format data by the java application we require JSON API.

Different vendors provided JSON API for the java language.

1. Jackson API
2. Gson (Google sun) API
3. JSON simple API

→ By using all the above API's from java application we can convert java object to json format and json format to java object.

JSON Example with Jackson API:



1)Jackson API

Provides Object Mapper class that is used to convert java obj into json and json into java obj.

ObjectMapper class important methods

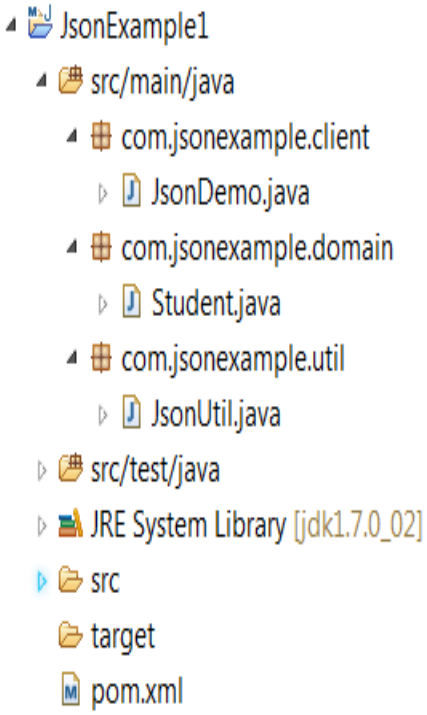
- public String writeValueAsString (-)-->used to convert java obj into json
- public T readValue(-)-->used to convert json into java obj
- JACKSON API is having two versions 1.x and 2.x versions
- if we are using Jackson 1.x version we can use the following jar's

- jackson-core-asl jar
- jackson-mapper-asl jar
- the same above jar's if we want get with maven we can add the following dependencies in pom.xml

```
<dependency>
        <groupId>org.codehaus.jackson</groupId>
        <artifactId>jackson-mapper-asl</artifactId>
        <version>1.9.3</version>
</dependency>
```

- if we are using jackson 2.x version we can following jar's
- jackson-core jar
- jackson-annotations jar
- jackson-databinding jar
- with maven if we want to get the jar' we can add following dependencies in pom.xml file

```
<dependency>
      <groupId>com.fasterxml.jackson.core</groupId>
      <artifactId>jackson-core</artifactId>
      <version>2.5.0</version>
</dependency>

<dependency>
      <groupId>com.fasterxml.jackson.core</groupId>
      <artifactId>jackson-annotations</artifactId>
      <version>2.5.0</version>
</dependency>

<dependency>
      <groupId>com.fasterxml.jackson.core</groupId>
      <artifactId>jackson-databind</artifactId>
      <version>2.5.0</version>
</dependency>
```



Student.java

```
package com.jsonexample.domain;
import java.io.Serializable;
public class Student implements Serializable{
    private Integer studentId;
    private String name;
    private String course;
    public Integer getStudentId() {
        return studentId;
    }
    public void setStudentId(Integer studentId) {
        this.studentId = studentId;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getCourse() {
        return course;
    }
    public void setCourse(String course) {
        this.course = course;
    }
}
```

JsonUtil.java

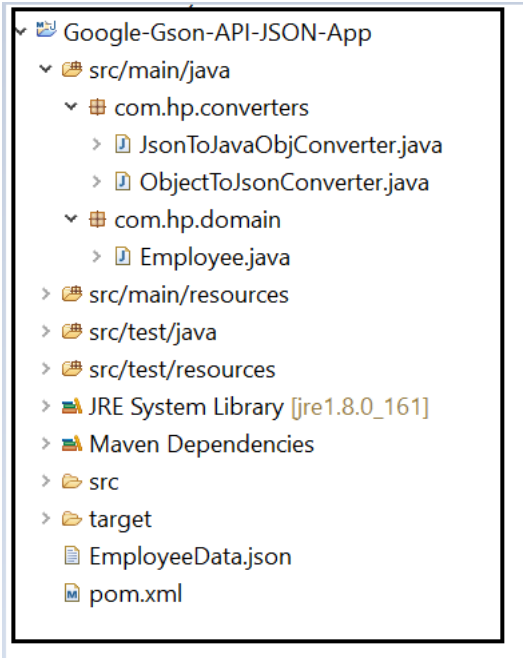
```java
package com.jsonexample.util;
import java.io.IOException;
public class JsonUtil {
public static String convertJavaToJson(Object obj){
    String json="{}";
ObjectMapper objectMapper=new ObjectMapper();
try{
json=objectMapper.writeValueAsString(obj);
}catch(JsonProcessingException e){
    e.printStackTrace();
}
return json;
}
public static <T> T convertJsonToJava(String jsonStr,Class<T> targetCls){
    T response=null;
    try {
    ObjectMapper objectMapper=new ObjectMapper();
    response=objectMapper.readValue(jsonStr,targetCls);
    } catch (JsonParseException e) {
        e.printStackTrace();
    } catch (JsonMappingException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
    return response;
}
}
```

JsonDemo.java

```java
package com.jsonexample.client;
import com.jsonexample.domain.Student;
import com.jsonexample.util.JsonUtil;
public class JsonDemo{
  public static void main( String[] args) {
   Student student=new Student();
   student.setStudentId(101);
   student.setName("raja");
   student.setCourse("java");
    String jsonStudent=JsonUtil.convertJavaToJson(student);
    System.out.println(jsonStudent);
 Student std1=JsonUtil.convertJsonToJava(jsonStudent,Student.class);
 System.out.println(std1.getStudentId()+" "+std1.getName()+" "+std1.getCourse());
    }
}
```

GSON Api to work with JSON Data

Create project like below



Add Gson maven dependency in pom.xml file like below

```
<dependencies>
    <dependency>
        <groupId>com.google.code.gson</groupId>
        <artifactId>gson</artifactId>
        <version>2.3.1</version>
    </dependency>
</dependencies>
```

Below is the model class to hold the data

Employee.java

```java
package com.hp.domain;

import com.google.gson.annotations.Expose;

public class Employee {

    @SerializedName("emp_id")
    @Expose
    private int empId;

    @Expose
    private String name;

    @Expose
    private String designation;

    private String department;

    @Expose
    private int salary;

    //setters & getters
    //toString( )

}
```

Below class is used to convert Object data to Json data

ObjectToJsonConverter.java

```java
package com.hp.converters;

import com.google.gson.Gson;

public class ObjectToJsonConverter {

    public static void main(String[] args) {

        Employee e = new Employee();
        e.setEmpId(101);
        e.setName("Ashok");
        e.setDesignation("Consultant");
        e.setDepartment("Development");
        e.setSalary(5000);

        //To enable Pretty Printing (Json format)
        //Ignoring the fields which does not have @Expose Annotation
        Gson gson = new GsonBuilder()
                    .setPrettyPrinting()
                    .excludeFieldsWithoutExposeAnnotation()
                    .create();
        String jsonString = gson.toJson(e);
        System.out.println(jsonString);
    }
}
```

Below file contains data in json format

*EmployeeData.json

```
{
  "emp_id": 101,
  "name": "Ashok",
  "designation": "Consultant",
  "department": "Development",
  "salary": 5000
}
```

Below class is used to convert Json data into Java object

JsonToJavaObjConverter.java

```
package com.hp.converters;

import java.io.FileReader;

public class JsonToJavaObjConverter {

    public static void main(String[] args) throws Exception {
        FileReader fr = new FileReader("EmployeeData.json");
        Gson gson = new Gson();
        Employee emp = gson.fromJson(fr, Employee.class);
        System.out.println(emp);
    }
}
```

Jersey implementation:

In order to simplify development of RESTful Web services and their clients in Java, a standard and portable JAX-RS API has been designed. Jersey RESTful Web Services implementation is an open source, production quality ,used for developing RESTful Web Services in Java that provides support for JAX-RS APIs and serves as a JAX-RS (JSR 311 & JSR 339) Reference Implementation.

Download the Jersey jar's from ->https://jersey.java.net/

- Jersey implementation is given by sun and it is having two versions 1.x and 2.x
- Jersey 1.x provided Servlet com.sun.jersey.spi.container.servlet.ServletContainer
- Jersey 2.x provided Servlet org.glassfish.jersey.servlet.ServletContainer

Steps to develop the Restful application:

step1: - create Root Resource class
step2: - create resource methods to expose
Step3: - configure the servlet in web.xml
(The Servlet is providing by JAX-RS API implementations)
step4: - deploy the app into server

Root Resource

- It is a server-side class which contains the data and functionality which is exposed as Resource class to the Client.
- *Root resource classes* are simple java classes that are annotated with @Path annotation.
- The Resource class methods are required to denote with resource method designator annotation such as @GET, @PUT, @POST, @DELETE.
- Resource methods are methods of a resource class annotated with a resource method designator annotation and exposing over the Network.

Root Resource class Example to produce JSON Response:

```
@Path("products")
public class ProductService{
@GET
@Path("getProduct")
@Produces(MediaType.APPLICATION_JSON)
// indicates the java method will produces content/response  in JSON format
        public Product searchProduct(){
            //logic
        }
}
```

- Any class that is annotated with @Path Annotation is called as root Resource class.
- In the Restfull services the Resources classes are associated with URI to identify and access it.
- The @Path annotation's value is a relative URI path. In the example above, the Java class will be hosted at the URI path /products.

Request Method Designator:

In the Resource class write the methods which we want to expose the over the Http protocols as resource methods .The Methods should be annotated with relevant Http method designators like @GET,@POST,@PUT and @DELETE.

- When we sent a GET request the method with @GET annotation would be executed to handle the request.

- @GET, @PUT, @POST, @DELETE and @HEAD are *resource method designator* annotations defined by JAX-RS and which correspond to the similarly named HTTP methods. In the example above, the annotated Java method will process HTTP GET requests.

Http Annotations

- **@javax.ws.rs.GET**
- **@javax.ws.rs.POST**
- **@javax.ws.rs.PUT**
- **@javax.ws.rs.DELETE**

Resource method:

- A method in a resource class that is annotated with Request Method Designator is called as Resource method.
- A lot of information we want pull from an HTTP request and injecting it into a Java method.
- You might be interested in a URI query string value.
- The client might be sending critical HTTP headers or cookie values that your service needs to process the request.
- JAX-RS lets you grab this information as you need it, through a set of injection annotations and APIs.

JAX-RS API Annotations

- **@javax.ws.rs.Path**
- **@javax.ws.rs.Consumes**
- **@javax.ws.rs.Produces**
- **@javax.ws.rs.QueryParam**
- **@javax.ws.rs.PathParam**
- **@javax.ws.rs.HeaderParam**
- **@javax.ws.rs.FormParam**
- **@javax.ws.rs.CookieParam**
- **@javax.ws.rs.MatrixParam**
- **@javax.ws.rs.BeanParam**

JAX-RS Annotations Overview

@QueryParam:

It is used for receiving the data as input through query String parameters as shown below.

```
@Path("products")
public class ProductResource {

@GET
@Produces(MediaType.APPLICATION_JSON)
@Path("searchProduct")
public Product  getProduct(@QueryParam("pid") Integer pid){

  if(pid!=null && pid.equals(101))     {
        Product product=new Product();
        product.setProductId(101);
        product.setProductName("mouse-101");
        product.setPrice(300.0);
        return product;
  }else{
        return null;
  }

}
```

ProductResource.java

- The client is the following URL to access the above Service
- **URL:-** http://<hostName>:<portNum>/context-root /products/searchProduct?pid=101

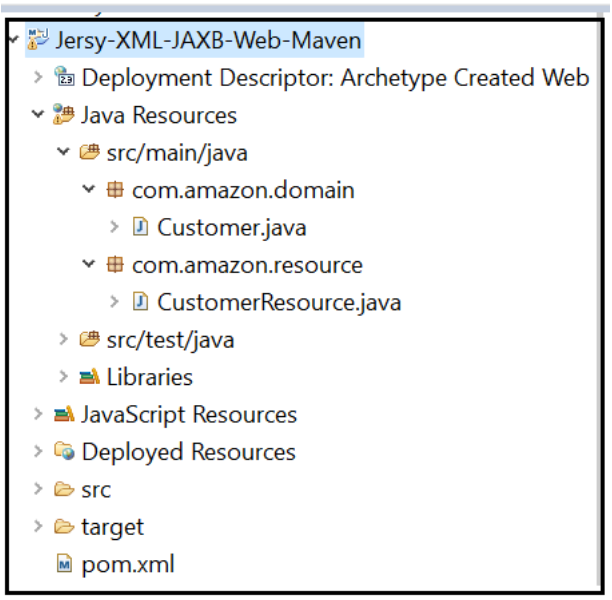
- The above URL shows the query param pid=101 where 101 value will be passed as an input to parameter pid of the method.
- The QueryParameters are name-value paris.
- The QueryParameters are starts with ? in the URL and if multiple Query parameters are there separated with & symbol

Ex2: Resource method with Multiple Query-Parameters

URL:http://<hostName>:<portNum>/context-root/products/searchProduct?pid=101&pname=mouse

Jersy 1.x Application

Step-1: - Create Maven Web application



Step-2: - Add Jersey Maven Dependencies in pom.xml file

```
<dependencies>
    <dependency>
        <groupId>com.sun.jersey</groupId>
        <artifactId>jersey-server</artifactId>
        <version>1.17</version>
    </dependency>
    <dependency>
        <groupId>com.sun.jersey</groupId>
        <artifactId>jersey-servlet</artifactId>
        <version>1.17</version>
    </dependency>
</dependencies>
```

Step-3: - Create a domain class (This is Binding class)

Customer.java

```
1  package com.amazon.domain;
2
3  import javax.xml.bind.annotation.XmlAccessType;
8
9  @XmlRootElement(name = "customer")
10 @XmlAccessorType (XmlAccessType.FIELD)
11 public class Customer {
12
13     @XmlAttribute(name = "customer-id")
14     private int customerId;
15
16     @XmlElement
17     private String customerName;
18
19     @XmlElement
20     private String customerEmail;
21
22     @XmlElement
23     private String customerPhno;
24
25     //setters & getters
26
27 }
```

Step-5: - Create Resource class

Step-6: - Configure Jersey Runtime Servlet in web.xml file like below

Step-7: - Deploy the application and test Rest resource using POSTMAN tool

@Matrix Param:

The idea of matrix parameters is that they are an arbitrary set of name-value pairs embedded in a uri path segment.

- The Matrix Parameters are starts with semicolon and if multiple parameters are there then separated with semicolon.
- The Matrix Parameters are Name-Value Pairs.

```
@Path("/car")
public class CarResource {

        @GET
        @Path("/order/{brand}/{model}")
        public String orderCar(
                @PathParam("brand") String brand,
                @PathParam("model") int model,
                @MatrixParam("color") String color
        ) {

                System.out.println("Brand : "+brand);
                System.out.println("Model : "+model);
                System.out.println("Color : "+color);
                return "Car orderd successfully..!";
        }
}
```

CarResource.java

Url to access car resource :

http://localhost:6060/FirstRestEasyWeb/rest/car/order/benz;color=red/2001

@FormParam

- **@FormParam to bind html form fields to your method inputs.**
- **It works for http method POST.**

```
<form action="rest/user-form/register" method="post">
      <p>Name : <input type="text" name="name" /></p>
      <p>Address : <input type="text" name="address" /></p>
      <input type="submit" value="Register" />
</form>
```
register.html

When register.html form got submitted, request comes to UserRegResource class registerUserInfo () method.

As part of this method we are capturing form data using @FormParam annotaion

```
@Path("/user-form")
public class UserRegResource {

    @POST
    @Path("/register")
    public Response registerUserInfo(@FormParam("name") String name,
             @FormParam("address") String address) {

        String response = "Successfully added user details, name: "+
                            name+" and address: "+address;

        return Response.status(200).entity(response).build();
    }
}
```
UserRegResource.java

@BeanParam

- The @BeanParam annotation is something new added in the JAX-RS 2.0 specification.
- It allows you to inject an application-specific class whose property methods or fields are annotated with any of the injection parameters

If we don't have @BeanForm → We need to write many @FormParam annotations to capture a form data. If form contains 20 fields then we have to 20 parameters for the method which is not recommended. Below image depicts the same.

```
/*
Just a sample method to show life without the @BeanParam annotation
*/
public Response withoutBeanParam1(@FormParam("") String isbn,
                                  @FormParam("") String name,
                                  @FormParam("") String author,
                                  @FormParam("") String publisher){

    //create a new Book object using the injected parameters

    //Book book = new Book(isbn, name, author, publisher);

    //save book

    return Response.ok(book.toString()).build();
}
```

To avoid the above problem, we can create a model class to store form data

```
@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Book {

    @FormParam("isbn")
    private String isbn;

    @FormParam("name")
    private String name;

    @FormParam("author")
    private String author;

    @FormParam("publisher")
    private String publisher;

    public Book() {
    }

    @Override

    public String toString(){
        return "Book [Name : " + name + "  ISBN : " + isbn + "  Author : "+ author +"  Publisher : "+ publisher +" ]";
    }
```

instance variables annotated with the @FormParam

We can use @BeanParam to inject form data into model/bean object like below

```
@POST
@Consumes({MediaType.APPLICATION_FORM_URLENCODED})

public Response createBook(@BeanParam Book book ){

    //save book

    Logger.getLogger("RESTResourceWithBeanParam").info(book.toString());

    return Response.ok(book.toString()).build();
}
```

JAX-RS provider automatically constructs and injects an instance of your domain object which you can now use within your methods.

Jersey Application with JSON Data

Step-1 → Create Maven Web Application

Step-2 → Add below maven dependencies in project pom.xml

```
<dependencies>
	<dependency>
		<groupId>com.sun.jersey</groupId>
		<artifactId>jersey-server</artifactId>
		<version>1.17</version>
	</dependency>
	<dependency>
		<groupId>com.sun.jersey</groupId>
		<artifactId>jersey-servlet</artifactId>
		<version>1.17</version>
	</dependency>
	<dependency>
		<groupId>com.sun.jersey</groupId>
		<artifactId>jersey-json</artifactId>
		<version>1.17</version>
	</dependency>
</dependencies>
```

Step-3 → Create a domain class to hold the data

```
public class Customer {

      @JsonProperty("customer-id")
      private int customerId;

      @JsonProperty("customer-name")
      private String customerName;

      @JsonProperty("customer-email")
      private String customerEmail;

      @JsonProperty("customer-phno")
      private String customerPhno;

      //setters & getters
      //toString ( ) method
}
```

Customer.java

Step-4 → Create Resource class with resource methods

The above Resource having a get request method and post request method

POST request method is used to add the customer using customer data in JSON format

GET request method is used to retrieve Customer data in JSON format

Step-5 →Create web.xml file with below details

- In the above web.xml file we can see two init parameters
- Jersey.config.server.provider.packages is used to load all resource classes before getting request (This will improve performance of the application). If we don't write this everytime resource class should be loaded from classpath.
- POJOMappingFeature init param is used to map json data to pojo and vice versa.

Step-6 → Deploy the application and test it using POSTMAN

To test the above Resource we can use SOAPUITool/PostMan/RestClient tool (OR) write a client application.

@Consumes

- The @Consumes annotation is used to specify which MIME media types of representations a resource can accept, or consume, from the client. If @Consumes is applied at the class level, all the response methods accept the specified MIME types by default. If @Consumes is applied at the method level, it overrides any @Consumes annotations applied at the class level.
- If a resource is unable to consume the MIME type of a client request, the Jersey runtime sends back an HTTP "415 Unsupported Media Type" error.
- The value of @Consumes is an array of String of acceptable MIME types. For example:
- @Consumes ({"text/plain, text/html"})

The following Example is showing how to consume the data in XML by using @Consumes annotation.

web.xml

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
  <servlet>
  <servlet-name>jerseyServlet</servlet-name>
  <servlet-class>
 com.sun.jersey.spi.container.servlet.ServletContainer
  </servlet-class>
  <load-on-startup>1</load-on-startup>
  </servlet>
<servlet-mapping>
<servlet-name>jerseyServlet</servlet-name>
<url-pattern>/*</url-pattern>
</servlet-mapping>
</web-app>
```

ProductService.java

ProductService.java

```
package com.nit.service;
import javax.ws.rs.Consumes;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
@Path("products")
public class ProductService {
    @POST
    @Path("/registerProduct")
    @Produces(MediaType.TEXT_PLAIN)
    @Consumes(MediaType.APPLICATION_XML)
public String
registerProduct( String productXML){
System.out.println("In server :");
System.out.println(productXML);
return "Product Registration Completed Successfully";
}
}
```

We can test above service by using the following client application As follows

```
package com.nit.client;
import java.io.StringWriter;
import javax.ws.rs.core.MediaType;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Marshaller;
import com.nit.domain.Product;
import com.sun.jersey.api.client.Client;
import com.sun.jersey.api.client.ClientResponse;
import com.sun.jersey.api.client.WebResource;
import com.sun.jersey.api.client.WebResource.Builder;
public class Test {
        public static void main(String [] args) throws JAXBException {
String URL = "http://hostName:portNum/contextpath/products/registerProduct";
                Client client = Client.create();
                WebResource webResource = client.resource(URL);
                Builder builder = webResource.type(MediaType.APPLICATION_XML);
                builder.accept(MediaType.TEXT_PLAIN);
                Product product = new Product();
                product.setPid(201);
                product.setPname("mouse");
                product.setPrice(900.0);
                JAXBContext jaxbContext = JAXBContext.newInstance(Product.class);
                Marshaller marshaller = jaxbContext.createMarshaller();
                StringWriter writer = newStringWriter();
                marshaller.marshal(product, writer);
                String productXML = writer.toString();
        ClientResponse clientResponse = builder.post(ClientResponse.class,productXML);
                String response = clientResponse.getEntity(String.class);
        s.o.p (clientResponse.getStatus()+""+ clientResponse.getStatusInfo());
        S.o.p (response);
        }
}
```

Product.java

```
package  com.nit.domain;
Import java.io.Serializable;
Import javax.xml.bind.annotation.XmlAttribute;
Import javax.xml.bind.annotation.XmlElement;
Import javax.xml.bind.annotation.XmlRootElement;
@XmlRootElement(name="product")
public class Product implements Serializable{
        private int pid;
        private String pname;
        private double price;
        @XmlAttribute(name="pid")
        public int getPid() {
                return pid;
        }
        public void setPid(int pid) {
                this.pid = pid;
        }
        @XmlElement
        public String getPname() {
                return pname;
        }
        public void setPname(String pname) {
                this.pname = pname;
        }
        @XmlElement
        public double getPrice() {
  return price;
        }
        Public void setPrice(double price) {
                this.price = price;
        }
}
```

Jersey 2.x Example

- Jersey_Book_Service_Util
  - src/main/java
    - com.jerseyutil.domain
      - Book.java
      - ResponseDTO.java
    - com.jerseyutil.util
      - JsonUtil.java
  - src/test/java
  - Maven Dependencies
  - JRE System Library [JavaSE-1.7]
  - src
  - target
  - pom.xml

```java
package com.jerseyutil.domain;
import java.io.Serializable;
public class Book implements Serializable{
    private String isbn;
    private String title;
    private String author;
    private Double price;
    public String getIsbn() {
        return isbn;
    }
    public void setIsbn(String isbn) {
        this.isbn = isbn;
    }
    public String getTitle() {
        return title;
    }
    public void setTitle(String title) {
        this.title = title;
    }
    public String getAuthor() {
        return author;
    }
    public void setAuthor(String author) {
        this.author = author;
    }
    public Double getPrice() {
        return price;
    }
```

ResponseDTO.java

```java
package com.jerseyutil.domain;
import java.io.Serializable;
public class ResponseDTO implements Serializable{
    private byte status;
    private String message,data;
    public byte getStatus() {
        return status;
    }
    public void setStatus(byte status) {
        this.status = status;
    }
    public String getMessage() {
        return message;
    }
    public void setMessage(String message) {
        this.message = message;
    }
    public String getData() {
        return data;
    }
    public void setData(String data) {
        this.data = data;
    }

}
```

JsonUtil.java

```java
import com.fasterxml.jackson.databind.JsonMappingException;
import com.fasterxml.jackson.databind.ObjectMapper;
public class JsonUtil {
public static String convertJavaToJson(Object obj){
String jsonStr="{}";
ObjectMapper objectMapper=new ObjectMapper();
  try {
    jsonStr=objectMapper.writeValueAsString(obj);
} catch (JsonProcessingException e) {
    e.printStackTrace();
}
  return jsonStr;
}
public static <T> T convertJsonToJava(String jsonStr,Class<T> cls){
    T response=null;
ObjectMapper objectMapper=new ObjectMapper();
   try {
    response=objectMapper.readValue(jsonStr,cls);
} catch (JsonParseException e) {
    e.printStackTrace();
} catch (JsonMappingException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}
    return response;
}
}
```

Jersey_Book_Service_Util/pom.xml

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
<groupId>com.jerseyutil</groupId>
  <artifactId>Jersey_Book_Service_Util</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>
    <name>Jersey_Book_Service_Util</name>
  <url>http://maven.apache.org</url>
<dependencies>
    <dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
   <artifactId>jackson-core</artifactId>
    <version>2.5.0</version>
    </dependency>
    <dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
   <artifactId>jackson-annotations</artifactId>
    <version>2.5.0</version>
    </dependency>
    <dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
   <artifactId>jackson-databind</artifactId>
    <version>2.5.0</version>
    </dependency>
  </dependencies>
</project>
```

- Jersey_Book_Service_Provider
  - src/main/java
    - com.jerseybookserviceprovider.dao
      - BookDAO.java
    - com.jerseybookserviceprovider.factory
      - ConnectionFactory.java
      - DAOFactory.java
    - com.jerseybookserviceprovider.service
      - BookResource.java
  - src/main/resources
  - src/test/java
  - Maven Dependencies
  - JRE System Library [JavaSE-1.7]
  - src
    - main
      - webapp
        - WEB-INF
          - web.xml
    - test
  - target
  - pom.xml

pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-
v4_0_0.xsd">
<modelVersion>4.0.0</modelVersion>
<groupId>com.jerseyserverexample</groupId>
<artifactId>Jersey_Book_Service_Provider</artifactId>
<packaging>war</packaging>
<version>1.0</version>
<name>Jersey_Book_Service_Provider MavenWebapp</name>
<url>http://maven.apache.org</url>
<dependencies>
<dependency>
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>3.8.1</version>
<scope>test</scope>
</dependency>
<dependency>
<groupId>org.glassfish.jersey.containers</groupId>
<artifactId>jersey-container-servlet</artifactId>
<version>2.22.2</version>
</dependency>
<dependency>
<groupId>javax.servlet</groupId>
<artifactId>javax.servlet-api</artifactId>
<version>3.0.1</version>
</dependency>
<!-- https://mvnrepository.com/artifact/com.hynnet/oracle-driver-ojdbc6 -->
<dependency>
   <groupId>com.hynnet</groupId>
   <artifactId>oracle-driver-ojdbc6</artifactId>
   <version>12.1.0.1</version>
</dependency>
<dependency>
<groupId>org.glassfish.jersey.media</groupId>
<artifactId>jersey-media-json-jackson</artifactId>
<version>2.22.2</version>
</dependency>
<dependency>
<groupId>com.jerseyutil</groupId>
<artifactId>Jersey_Book_Service_Util</artifactId>
<version>0.0.1-SNAPSHOT</version>
</dependency>
</dependencies>
<build>
<finalName>Jersey_Book_Service_Provider</finalName>
</build>
</project>
```

web.xml

```
<web-app>
  <servlet>
    <servlet-name>Jersy</servlet-name>
    <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
    <init-param>
   <param-name>
   jersey.config.server.provider.packages</param-name>
   <param-value>
   com.jerseybookserviceprovider.service
 </param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>Jersy</servlet-name>
    <url-pattern>/*</url-pattern>
  </servlet-mapping>
</web-app>
```

BookResource.java

```
@Path("books")
public class BookResource {
@POST
@Produces(MediaType.APPLICATION_JSON)
@Path("/registerBook")
@Consumes(MediaType.APPLICATION_JSON)
public ResponseDTO
registerBook(String jsonBook){
    System.out.println("Server :"+jsonBook);
    ResponseDTO response=new ResponseDTO();
    response.setMessage("Server Problem ,Book Details Could not be received ");
    if(jsonBook!=null){
Book book= JsonUtil.convertJsonToJava(jsonBook,Book.class);
int count=DAOFactory.getBookDAO().registerBook(book);
if(count>0){
    response.setStatus((byte)1);
    response.setData(book.getIsbn());
    response.setMessage("Book Details saved Successfully");
}
else{
    response.setMessage("Book Details could not be saved,Please Try Again!");
}
System.out.println("Server Response :"+response.getMessage());
    }
    return response;
}
}
```

BookDAO.java

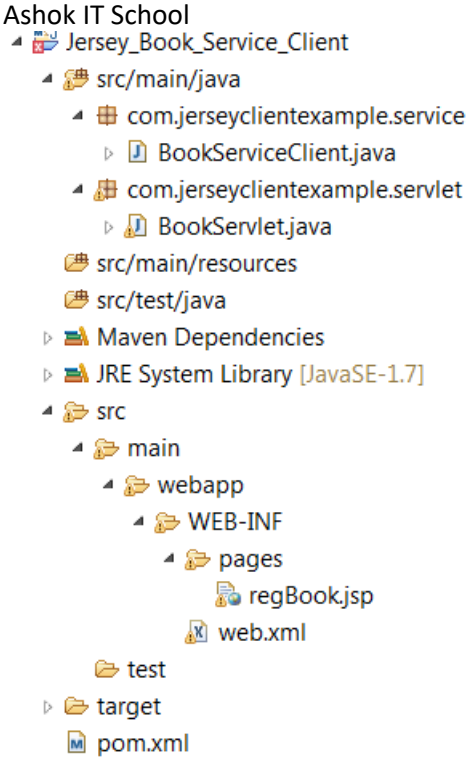
```
package com.jerseybookserviceprovider.dao;
import java.sql.Connection;
public class BookDAO {
private static final String SQL_REGISTER_BOOK="insert into Book_Details values(?,?,?,?)";
    public int registerBook(Book book){
        int count=0;
try {
Connection con=ConnectionFactory.getConnection();
PreparedStatement pst=con.prepareStatement(SQL_REGISTER_BOOK);
pst.setString(1,book.getIsbn());
pst.setString(2,book.getTitle());
pst.setString(3,book.getAuthor());
pst.setDouble(4,book.getPrice());
  count=pst.executeUpdate();
} catch (SQLException e) {
    e.printStackTrace();
}
return count;
}
}
```

DAOFactory.java

```
package com.jerseybookserviceprovider.factory;
import com.jerseybookserviceprovider.dao.BookDAO;
public class DAOFactory {
private static BookDAO bookDAO;
static{
    bookDAO=new BookDAO();
}
public static BookDAO getBookDAO(){
    return bookDAO;
}
}
```

ConnectionFactory.java

```
package com.jerseybookserviceprovider.factory;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
public class ConnectionFactory {
static{
try {
    Class.forName("oracle.jdbc.driver.OracleDriver");
} catch (ClassNotFoundException e) {
System.out.println("Exception Occured while loading the driver class :"+e.getMessage());
}
}
public static Connection getConnection() throws SQLException{
    String url="jdbc:oracle:thin:@localhost:1521:XE";
    String un="system";
    String pass="manager";
    Connection con=DriverManager.getConnection(url,un,pass);
    return con;
}
}
```



pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
<modelVersion>4.0.0</modelVersion>
<groupId>com.jerseyclientexample</groupId>
<artifactId>Jersey_Book_Service_Client</artifactId>
<packaging>war</packaging>
<version>0.0.1-SNAPSHOT</version>
<name>Jersey_Book_Service_Client MavenWebapp</name>
<url>http://maven.apache.org</url>
<dependencies>
<dependency>
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>3.8.1</version>
<scope>test</scope>
</dependency>
<dependency>
<groupId>org.glassfish.jersey.core</groupId>
<artifactId>jersey-client</artifactId>
<version>2.22.2</version>
</dependency>
<dependency>
<groupId>org.glassfish.jersey.media</groupId>
<artifactId>jersey-media-json-jackson</artifactId>
<version>2.22.2</version>
```

```
</dependency>
<dependency>
<groupId>javax.servlet</groupId>
<artifactId>javax.servlet-api</artifactId>
<version>3.0.1</version>
</dependency>
<dependency>
<groupId>com.jerseyutil</groupId>
<artifactId>Jersey_Book_Service_Util</artifactId>
<version>0.0.1-SNAPSHOT</version>
</dependency>
</dependencies>
<build>
<finalName>Jersey_Book_Service_Client</finalName>
</build>
</project>
```

regBook.jsp

```
<html>
<%@page isELIgnored="false" %>
<head><h2 align="center">Book Registration</h2></head>
<br/>${responseDTO.message <hr/>
<body>
<form action="saveBookDetails" method="post">
Isbn :<input type="text"  name="isbn"/><br/>
Title :<input type="text" name="title"/><br/>
Author :<input type="text" name="author"/><br/>
Price :<input type="text" name="price"/><br/>
<input type="submit" value="register"/>
</form>
</body>
</html>
```

web.xml

```
<!DOCTYPE web-app PUBLIC
 "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
 "http://java.sun.com/dtd/web-app_2_3.dtd" >
<web-app>
  <servlet>
  <servlet-name>bookServlet</servlet-name>
  <servlet-class>com.jerseyclientexample.servlet.BookServlet</servlet-class>
  <load-on-startup>2</load-on-startup>

  </servlet>
  <servlet-mapping>
   <servlet-name>bookServlet</servlet-name>
 <url-pattern>/saveBookDetails</url-pattern>
  </servlet-mapping>
  <welcome-file-list>
  <welcome-file>
  /WEB-INF/pages/regBook.jsp
  </welcome-file>
  </welcome-file-list>
</web-app>
```

BookServlet.java

```java
package com.jerseyclientexample.servlet;
import java.io.IOException;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.jerseyclientexample.service.BookServiceClient;
import com.jerseyutil.domain.Book;
import com.jerseyutil.domain.ResponseDTO;
public class BookServlet extends HttpServlet {
    private BookServiceClient bookServiceClient;
    public void init(){
        bookServiceClient=new BookServiceClient();
    }
public void doPost(HttpServletRequest req,HttpServletResponse res)throws ServletException,IOException{
    Book book=new Book();
    book.setIsbn(req.getParameter("isbn"));
    book.setTitle(req.getParameter("title"));
    book.setAuthor(req.getParameter("author"));
    book.setPrice(Double.parseDouble(req.getParameter("price")));
    ResponseDTO responseDTO=bookServiceClient.saveBookDetails(book);
    req.setAttribute("responseDTO",responseDTO);
    String target="/WEB-INF/pages/regBook.jsp";
    RequestDispatcher rd=req.getRequestDispatcher(target);
rd.forward(req, res);
}
}
```

Writable Smart Insert

BookServiceClient.java

```java
package com.jerseyclientexample.service;
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Entity;
import javax.ws.rs.client.Invocation.Builder;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import org.glassfish.jersey.client.ClientConfig;
import org.glassfish.jersey.filter.LoggingFilter;
import com.jerseyutil.domain.Book;
import com.jerseyutil.domain.ResponseDTO;
import com.jerseyutil.util.JsonUtil;
public class BookServiceClient {
public ResponseDTO saveBookDetails(Book book){
    String URL="http://localhost:8082/Jersey_Book_Service_Provider/books/registerBook";
    /*Jersey 2.x Client Code*/
Client client = ClientBuilder.newClient( new ClientConfig().register( LoggingFilter.class) );
    WebTarget target=client.target(URL);
      Builder builder=target.request(MediaType.APPLICATION_JSON);
      builder.accept(MediaType.APPLICATION_JSON);
Response clientResponse=builder.post(Entity.entity(book,MediaType.APPLICATION_JSON),Response.class);
       String jsonResponse=clientResponse.readEntity(String.class);
    ResponseDTO responseDTO=JsonUtil.convertJsonToJava(jsonResponse,ResponseDTO.class);
  return responseDTO;
}
}
```

The client obj is heavyweight object . It is always recomnded to close the client object after consuming the resource.

The client obj Establish the connection b.w client app to server app.

If we are using Jersey 1.x we can use the following code in the place of above code

jersey 1.x version client code

```
public class BookServiceClient{
public ResponseDTO saveBookDetails(Book book){
StringURL="http://localhost:8082/Jersey_Book_Service_Provider/books/registerBook";
Client client  = Client.create();
  WebResource resource=client.resource(URL);
Builder builder  =resource.accept(MediaType.APPLICATION_JSON);
    builder.type(MediaType.APPLICATION_JSON);
    ClientResponse
clientResponse=builder.post(ClientResponse.class,JsonUtil.convertJavaToJson(book));
      String jsonResponse=clientResponse.getEntity(String.class);
      ResponseDTO
responseDTO=JsonUtil.convertJsonToJava(jsonResponse,ResponseDTO.class);
return responseDTO;
}
}
```

@FormParam :

In JAX-RS, you can use @FormParam annotation to bind HTML form parameters value to a Java method. The following example show you how to do it:

Note: @FormParam is used to bind html form fields to your method inputs. It works only for http method POST.

First create a HTML page with Form Parameters

a simple HTML form with "**post**" method.

UserForm.html

```
<html>
<body>
<h1>JAX-RS @FormQuery Testing</h1>
<form action="user/register" method="post">
Name : <input type="text" name="name" />
Age : <input type="text" name="age" />
<input type="submit" value="Add User" />
</form>
</body>
</html>
```

Create A Resource class to get Form Parameters

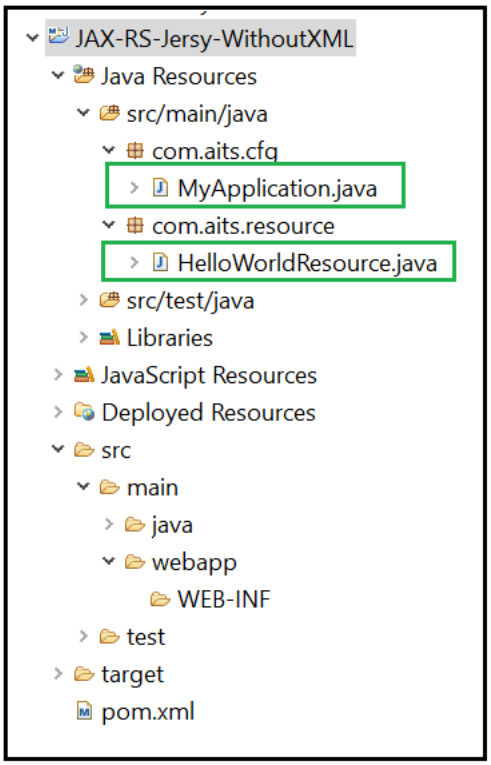
```
Example to use @FormParam and to get above HTML form parameter values.
import javax.ws.rs.FormParam;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.core.Response;
@Path("user")
public class UserService {

        @POST
        @Path("/register")
         @Produces("text/plian")
        public String addUser( @FormParam("name") String name,
                @FormParam("age") int age) {

                 return "addUser is called, name : " + name + ", age : " + age);
        }
}
```

Note:

Access HTML Page. URL: ***http://localhost:8080/RESTfulExample/UserForm.html***
Then click on Add User Button get the Output.

Restful Resource Development without XML configuration

Step-1 → Create Maven web project



Note: In the above application no web.xml file (delete web.xml file from WEB-INF folder)

Step-2 →Configure jersey dependencies in project pom.xml file

```
<properties>
      <failOnMissingWebXml>false</failOnMissingWebXml>
      <jersey.version>2.20</jersey.version>
</properties>

<dependencies>
      <dependency>
            <groupId>org.glassfish.jersey.containers</groupId>
            <artifactId>jersey-container-servlet</artifactId>
            <version>${jersey.version}</version>
      </dependency>
</dependencies>
```

pom.xml

Note : As we have deleted web.xml file we are writing <failOnMissingWebXml> propert to false so that it will not check for web.xml file

Step-3 → Create Application Configuration class (Extends from Application class)

Step-4 → Create a Resource class

```
package com.aits.resource;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

@Path("/hello")
public class HelloWorldResource {

	@GET
	@Produces({ MediaType.TEXT_PLAIN })
	public String getPlain() {
		return "Hello World!!!";
	}
}
```

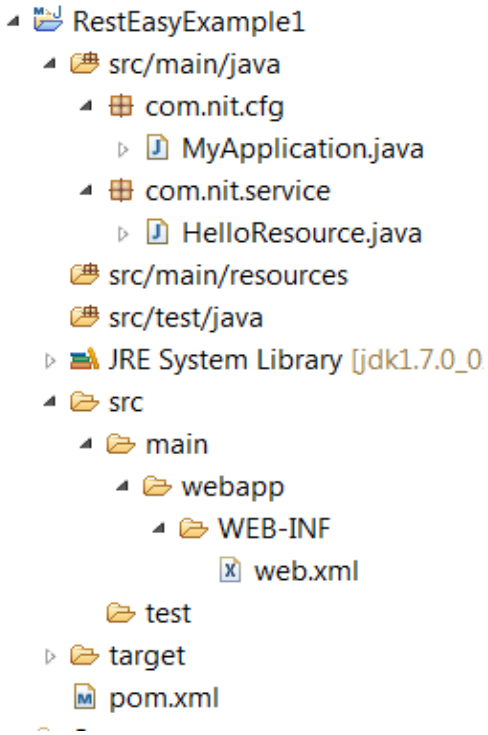
HelloWorldResource.java

Step-5 → Deploy the application and test it using POSTMAN tool

RestEasy Implementation:

- REST Easy is a JBoss project that provides various frameworks to help you build RESTful Web Services and RESTful Java applications. It is a **fully certified** and portable implementation of the JAX-RS 2.0 specification, a JCP specification that provides a Java API for RESTful Web Services over the HTTP protocol. The REST Easy can run in any Servlet container

- We can download the Rest Easy jar's from http://resteasy.jboss.org/downloads



HelloResource.java

```
package com.nit.service;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
@Path("hello")
public class HelloResource {
	@Path("/sayHello/{name}")
	@Produces(MediaType.TEXT_PLAIN)
	@GET
	public String sayHello(@PathParam("name") String name){
		return "Hello"+name+"Welcome to RestEasy";
	}
}
```

MyApplication.java

```
import java.util.HashMap;
import java.util.HashSet;
import java.util.Set;
importjavax.ws.rs.core.Application;
import com.nit.service.HelloResource;

public class MyApplication extends Application{

      private static Set<Object>set=new HashSet<Object>();

      public MyApplication() {
            HelloResource helloResource=new HelloResource();
            set.add(helloResource);
      }

      public Set<Object> getSingletons(){ //overriding method
            returnset;
      }
}
```

web.xml

```
<web-app>
 <servlet>
 <servlet-name>dispatcher</servlet-name>
 <servlet-class>org.jboss.resteasy.plugins.server.servlet.HttpServletDispatcher</servlet-class>
 <init-param>
 <param-name>javax.ws.rs.core.Application</param-name>
<param-value>com.nit.cfg.MyApplication</param-value>
 </init-param>
 </servlet>
 <servlet-mapping>
 <servlet-name>dispatcher</servlet-name>
 <url-pattern>/nit/*</url-pattern>
 </servlet-mapping>
  <context-param>
<param-name>resteasy.servlet.mapping.prefix</param-name>
<param-value>/nit</param-value>
</context-param>
</web-app>

<!-- when the URL-Pattern is except /* then we can configure
<context-parameter> in web.xml file to specify the prefix
 -->
```

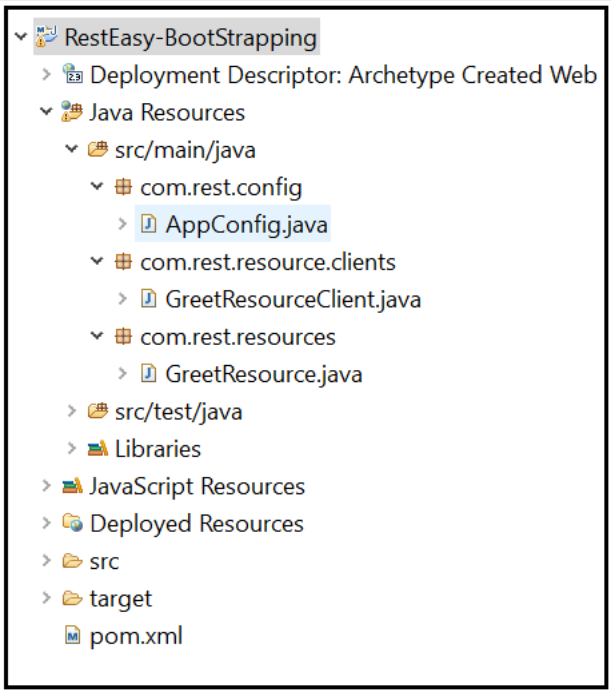
We can access the above webservice by using a client Application (OR) SOAP UI tool

RestEasy Client Application

Test.java

RestEasy-Bootstrapping Application

Step-1 → Create Maven Web application



Step-2 → Configure resteasy dependencies in project pom.xml file like below

```
<dependencies>
    <dependency>
        <groupId>org.jboss.resteasy</groupId>
        <artifactId>resteasy-jaxrs</artifactId>
        <version>2.2.1.GA</version>
    </dependency>
</dependencies>
```

Step-3 → Create Application Configuration class like below

AppConfig.java

```
1  package com.rest.config;
2
3  import java.util.HashSet;
9
10 public class AppConfig extends Application {
11
12     @Override
13     public Set<Object> getSingletons() {
14         Set<Object> singletons = new HashSet<Object>();
15         singletons.add(new GreetResource());
16         return singletons;
17     }
18
19     @Override
20     public Set<Class<?>> getClasses() {
21         return super.getClasses();
22     }
23 }
24
```

AppConfig.java

Step-4 → Create Resource class like below

GreetResource.java

```
1  package com.rest.resources;
2
3  import javax.ws.rs.GET;
5
6  @Path("/greet")
7  public class GreetResource {
8
9      @GET
10     public String greet() {
11         return "Good Morning...!";
12     }
13 }
14
```

GreetResource.java

Step-5 → Create web.xml file like below

<u>Below is the Client class to rest the above Rest Resource</u>

File Uploading with Restful Resource

Below is the Resource class to upload the file

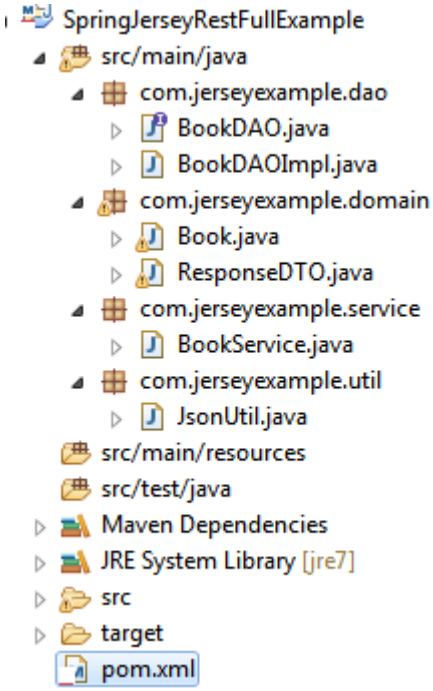
Postman screen shot to upload file (We can test the above FileUploadResource using postman)

To upload file from user interface we can use below html form

```
<form action="rest/files/upload" method="post"
            enctype="multipart/form-data">
            <p>
                  Select a file : <input type="file" name="file" />
            </p>
            <input type="submit" value="Upload File" />
</form>
```

index.html

Spring+RestFul services (Jersey Implementation) Integration Example




```
<projectxmlns="http://maven.apache.org/POM/4.0.0"xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0    http://maven.apache.org/maven-v4_0_0.xsd">
<modelVersion>4.0.0</modelVersion>
<groupId>com.springjerseyexample</groupId>
<artifactId>SpringJerseyRestFullExample</artifactId>
<packaging>war</packaging>
<version>0.0.1-SNAPSHOT</version>
<name>SpringJerseyRestFullExample MavenWebapp</name>
<url>http://maven.apache.org</url>
```

```
<dependencies>
<dependency>
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>3.8.1</version>
<scope>test</scope>
</dependency>
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-web</artifactId>
<version>3.0.5.RELEASE</version>
</dependency>
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-jdbc</artifactId>
<version>3.0.5.RELEASE</version>
</dependency>
<dependency>
<groupId>mysql</groupId>
<artifactId>mysql-connector-java</artifactId>
<version>5.1.35</version>
</dependency>
<!-- https://mvnrepository.com/artifact/com.sun.jersey/jersey-server -->
<dependency>
<groupId>com.sun.jersey</groupId>
<artifactId>jersey-server</artifactId>
<version>1.8</version>
</dependency>
<dependency>
<groupId>org.codehaus.jackson</groupId>
<artifactId>jackson-mapper-asl</artifactId>
<version>1.9.13</version>
</dependency>
<dependency>
                <groupId>com.sun.jersey.contribs</groupId>
                <artifactId>jersey-spring</artifactId>
                <version>1.8</version>
                <exclusions>
                        <exclusion>
                                <groupId>org.springframework</groupId>
                                <artifactId>spring</artifactId>
                        </exclusion>
                        <exclusion>
                                <groupId>org.springframework</groupId>
                                <artifactId>spring-core</artifactId>
                        </exclusion>
                        <exclusion>
                                <groupId>org.springframework</groupId>
```

```
                        <artifactId>spring-web</artifactId>
                    </exclusion>
                    <exclusion>
                        <groupId>org.springframework</groupId>
                        <artifactId>spring-aop</artifactId>
                    </exclusion>
                    <exclusion>
                        <groupId>org.springframework</groupId>
                        <artifactId>spring-beans</artifactId>
                    </exclusion>
                    <exclusion>
                        <groupId>org.springframework</groupId>
                        <artifactId>spring-context</artifactId>
                    </exclusion>
                </exclusions>
            </dependency>

</dependencies>
<build>
<finalName>SpringJerseyRestFullExample</finalName>
</build>
</project>
```

Book.java

```
package com.jerseyexample.domain;
import java.io.Serializable;
public class Book_implements Serializable {
private Integer isbn;
private String title,author,publication;
private Double price;
public Integer getIsbn() {
        returnisbn;
}
public void setIsbn(Integer isbn) {
        this.isbn = isbn;
}
public String getTitle() {
        returntitle;
}
public void setTitle(String  title) {
        this.title = title;
}
public String getAuthor() {
        return author;
}
public void setAuthor(String author) {
```

```
        this.author = author;
}
public String getPublication() {
        return publication;
}
public void setPublication(String publication) {
        this.publication = publication;
}
public Double getPrice() {
        return price;
}
public void setPrice(Double price) {
        this.price = price;
}

}
```

ResponseDTO.java

```
package  com.jerseyexample.domain;
import  java.io.Serializable;
public class ResponseDTO  implements Serializable{
private byte status;
private String msg;
private String data;
public byte getStatus() {
        return status;
}
public void setStatus(byte status) {
        this.status = status;
}
public String getMsg() {
        return msg;
}
public void setMsg(String msg) {
        this.msg = msg;
}
public String getData() {
        return data;
}
public void setData(String data) {
        this.data = data;
}
}
```

BookDAO.java

```
package  com.jerseyexample.dao;
import  com.jerseyexample.domain.Book;
public interface BookDAO {
public int registerBook(Book book);
}
```

BookDAOImpl.java

```
package com.jerseyexample.dao;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import javax.sql.DataSource;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;
import com.jerseyexample.domain.Book;
@Repository
public class BookDAOImpl implements BookDAO{
        @Autowired
        private DataSource dataSource;
  public int registerBook(Book book) {
        int count=0;
         Connection con=null;
        try{
con=dataSource.getConnection();
String sql="insert into Book_Details values(?,?,?,?,?)";
PreparedStatement pst=con.prepareStatement(sql);
pst.setInt(1,book.getIsbn());
pst.setString(2,book.getTitle());
pst.setDouble(3,book.getPrice());
pst.setString(4,book.getAuthor());
pst.setString(5,book.getPublication());
count=pst.executeUpdate();
         }catch(SQLException se){
                se.printStackTrace();
         }finally{
        if(con!=null){
                try{
            con.close();
                }catch(SQLException se){
                se.printStackTrace();
                }
        }
         }
                return count;
        }
}
```

BookService.java

```
package com.jerseyexample.service;
import javax.ws.rs.Consumes;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.jerseyexample.dao.BookDAO;
import com.jerseyexample.domain.Book;
import com.jerseyexample.domain.ResponseDTO;
import com.jerseyexample.util.JsonUtil;
@Path("books")
@Service
publicclass BookService{
        @Autowired
private BookDAO bookDAO;
@POST
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
@Path("/registerBook")
public String registerBook(String jsonBook){
        System.out.println("Entered into registerBook ::"+jsonBook);
Book book=JsonUtil.convertJsonToJava(Book.class,jsonBook);
String msg="Registration Is Failure!Please Try Again";
ResponseDTO responseDTO=new ResponseDTO();
responseDTO.setMsg(msg);
int count=bookDAO.registerBook(book);
if(count>0){
    msg="Registration Is Success";
        responseDTO.setMsg(msg);
        responseDTO.setStatus((byte)1);
        responseDTO.setData(book.getIsbn().toString());
  }
 System.out.println("Response of  registerBook ::"+responseDTO.getMsg());
 String  jsonResponseDTO=JsonUtil.convertJavaToJson(responseDTO);
return jsonResponseDTO;
}
}
```

JsonUtil.java

```
package com.jerseyexample.util;
import java.io.IOException;
import org.codehaus.jackson.JsonGenerationException;
import org.codehaus.jackson.JsonParseException;
import org.codehaus.jackson.map.JsonMappingException;
import org.codehaus.jackson.map.ObjectMapper;
public class JsonUtil {
public static <T> T convertJsonToJava(Class<T> cls,String  json){
	T response=null;
	ObjectMapper mapper=new ObjectMapper();
	try {
		response=mapper.readValue(json,cls);
	} catch (JsonParseException e) {
		e.printStackTrace();
	} catch (JsonMappingException e) {
		e.printStackTrace();
	} catch (IOException e) {
		e.printStackTrace();
	}
return response;
}
public static String convertJavaToJson(Object obj){
	String jsonStr="";
	ObjectMapper mapper=new ObjectMapper();
	try {
		jsonStr=mapper.writeValueAsString(obj);
	} catch (JsonGenerationException e) {
		e.printStackTrace();
	} catch (JsonMappingException e) {
		e.printStackTrace();
	} catch (IOException e) {
		e.printStackTrace();
	}
	return jsonStr;
}
}
```

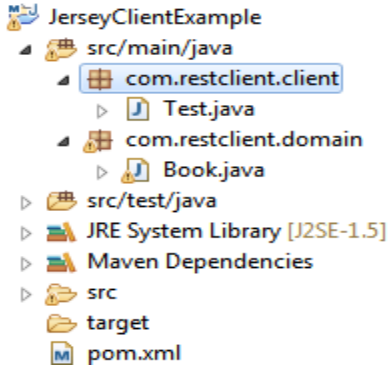
web.xml

```
<web-app>
<listener>
<listener-class>
org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
<servlet>
<servlet-name>jersey-serlvet</servlet-name>
<servlet-class>
com.sun.jersey.spi.spring.container.servlet.SpringServlet</servlet-class>
<init-param>
<param-name>com.sun.jersey.config.property.packages</param-name>
<param-value>com.jerseyexample.service</param-value>
</init-param>
<load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
<servlet-name>jersey-serlvet</servlet-name>
<url-pattern>/*</url-pattern>
</servlet-mapping>
</web-app>
```

applicationContext.xml

```
<beansxmlns="http://www.springframework.org/schema/beans"
xmlns:context="http://www.springframework.org/schema/context"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd">
<beanid="dataSource"class="org.springframework.jdbc.datasource.DriverManagerD
ataSource">
<propertyname="driverClassName"value="com.mysql.jdbc.Driver"/>
<propertyname="url"value="jdbc:mysql://localhost:3306/nit"/>
<propertyname="username"value="root"/>
<propertyname="password"value="root"/>
</bean>
<context:component-scanbase-
package="com.jerseyexample.service,com.jerseyexample.dao"/>
</beans>
```

The Following Client Application is used to Test Above Restful Service



pom.xml

```
<projectxmlns="http://maven.apache.org/POM/4.0.0"xmlns:xsi="http://www.w3.org/
2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>
<groupId>com.restclient.client</groupId>
<artifactId>JerseyClientExample</artifactId>
<version>0.0.1-SNAPSHOT</version>
<packaging>jar</packaging>
<name> JerseyClientExample </name>
<url>http://maven.apache.org</url>
<dependencies>
<dependency>
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>3.8.1</version>
<scope>test</scope>
</dependency>
<dependency>
<groupId>com.sun.jersey</groupId>
<artifactId>jersey-client</artifactId>
<version>1.19.1</version>
</dependency>
<dependency>
<groupId>org.codehaus.jackson</groupId>
<artifactId>jackson-mapper-asl</artifactId>
<version>1.9.13</version>
</dependency>
</dependencies>
</project>
```

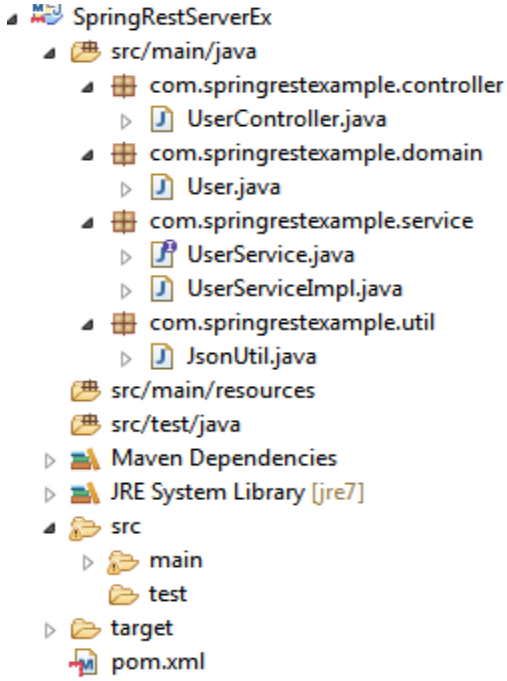
Book.java

```
package com.restclient.domain;
import java.io.Serializable;
public class Book implements Serializable {
private Integer isbn;
private String title,author,publication;
private Double price;
public Integer getIsbn() {
	return isbn;
}
public void setIsbn(Integer isbn) {
	this.isbn = isbn;
}
public String getTitle() {
	return title;
}
public void setTitle(String title) {
	this.title = title;
}
public String getAuthor() {
	return author;
}
public void setAuthor(String author) {
	this.author = author;
}
public String getPublication() {
	return publication;
}
public void setPublication(String publication) {
	this.publication = publication;
}
public Double getPrice() {
	return price;
}
public void setPrice(Double price) {
	this.price = price;
}
}
```

Test.java

```
package com.restclient.client;
import java.io.IOException;
import javax.ws.rs.core.MediaType;
import org.codehaus.jackson.JsonGenerationException;
import org.codehaus.jackson.map.JsonMappingException;
import org.codehaus.jackson.map.ObjectMapper;
import com.restclient.domain.Book;
import com.sun.jersey.api.client.Client;
import com.sun.jersey.api.client.ClientResponse;
import com.sun.jersey.api.client.WebResource;
import com.sun.jersey.api.client.WebResource.Builder;
public class Test{
    public static void main( String[] args ) throws JsonGenerationException,
JsonMappingException, IOException {
    String URL="http://localhost:8090/SpringJerseyRestFullExample/books/registerBook";
   Book book=new Book();
   book.setIsbn(302);
   book.setTitle("java");
   book.setPrice(1900.0);
   book.setPublication("nit");
   book.setAuthor("jhon");
   ObjectMapper mapper=new ObjectMapper();
   String jsonBook=mapper.writeValueAsString(book);
   System.out.println(jsonBook);
   Client client=Client.create();
   WebResource resource=client.resource(URL);
   Builder builder=resource.accept(MediaType.APPLICATION_JSON);
   builder.type(MediaType.APPLICATION_JSON);
  ClientResponse clientResponse=builder.post(ClientResponse.class,jsonBook);
   System.out.println(clientResponse.getStatus()+" "+clientResponse.getStatusInfo());
System.out.println(clientResponse.getEntity(String.class));
    }
}
```

Spring Rest-API Example:



pom.xml

```
<projectxmlns="http://maven.apache.org/POM/4.0.0"xmlns:xsi="http://www.w3.org/2001
/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
<modelVersion>4.0.0</modelVersion>
<groupId>com.springrestexample</groupId>
<artifactId>SpringRestExample</artifactId>
<packaging>war</packaging>
<version>0.0.1-SNAPSHOT</version>
<name>SpringRestExample MavenWebapp</name>
<url>http://maven.apache.org</url>
<dependencies>
<dependency>
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>3.8.1</version>
<scope>test</scope>
</dependency>
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-webmvc</artifactId>
<version>4.0.5.RELEASE</version>
</dependency>
<dependency>
<groupId>javax.servlet</groupId>
<artifactId>javax.servlet-api</artifactId>
```

```
<version>3.0.1</version>
</dependency>
<dependency>
<groupId>org.codehaus.jackson</groupId>
<artifactId>jackson-mapper-asl</artifactId>
<version>1.9.13</version>
</dependency>
</dependencies>
<build>
<finalName>SpringRestExample</finalName>
</build>
</project>
```

User.java

```
package com.springrestexample.domain;
publicclass User {
privateintuserId;
private String userName;
private String email,mobile;
publicint getUserId() {
	returnuserId;
}
publicvoid setUserId(intuserId) {
	this.userId = userId;
}
public String getUserName() {
	returnuserName;
}
publicvoid setUserName(String userName) {
	this.userName = userName;
}
public String getEmail() {
	returnemail;
}
publicvoid setEmail(String email) {
	this.email = email;
}
public String getMobile() {
	returnmobile;
}
publicvoid setMobile(String mobile) {
	this.mobile = mobile;
}
}
```

UserService.java

```
package com.springrestexample.service;
publicinterface UserService {
public String getUserDetails(Integer userId);
}
```

UserServiceImpl.java

```
package com.springrestexample.service;
import org.springframework.stereotype.Service;
import com.springrestexample.domain.User;
import com.springrestexample.util.JsonUtil;
@Service
publicclass UserServiceImpl implements UserService{
public String getUserDetails(Integer userId) {
	String jsonUser="{}";
	if(userId!=null&&userId.equals(101)){
		User user=new User();
		user.setUserId(101);
		user.setUserName("rama");
		user.setEmail("rama@gmail.com");
		user.setMobile("999999");
	jsonUser=JsonUtil.javaToJson(user);
		}
		returnjsonUser;
	}
}
```

JsonUtil.java

```
package com.springrestexample.util;
import java.io.IOException;
import org.codehaus.jackson.JsonGenerationException;
import org.codehaus.jackson.JsonParseException;
import org.codehaus.jackson.map.JsonMappingException;
import org.codehaus.jackson.map.ObjectMapper;
publicclass JsonUtil {
publicstatic String javaToJson(Object obj){
	String jsonStr="{}";
	ObjectMapper mapper=new ObjectMapper();
	try {
		jsonStr=mapper.writeValueAsString(obj);
	} catch (JsonGenerationException e) {
		e.printStackTrace();
	} catch (JsonMappingException e) {
		e.printStackTrace();
	} catch (IOException e) {
		e.printStackTrace();
	}

	returnjsonStr;
}
publicstatic<T>T jsonToJava(Class<T>cls,String str){
	T response=null;
ObjectMapper mapper=new ObjectMapper();
	try {
		response=mapper.readValue(str,cls);
	} catch (JsonParseException e) {
		e.printStackTrace();
	} catch (JsonMappingException e) {
		e.printStackTrace();
	} catch (IOException e) {
		e.printStackTrace();
	}
	returnresponse;
}
}
```

UserController.java

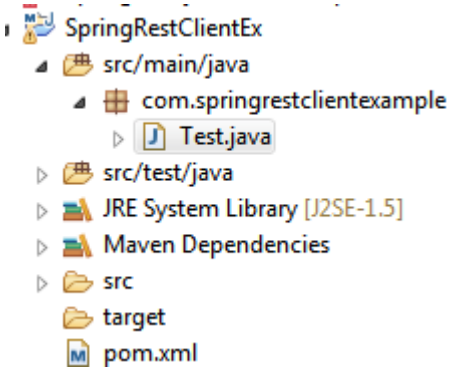
```
package com.springrestexample.controller;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;
import com.springrestexample.service.UserService;
@RestController
publicclass UserController {
        @Autowired
        private UserService userService;
@RequestMapping(value="getUserInfo/{userId}",method=RequestMethod.GET)
@ResponseBody
public String getUserDetails(@PathVariable("userId") Integer userId){
         String jsonUser=userService.getUserDetails(userId);
        returnjsonUser;
}
}
```

web.xml

```
<web-app>
<servlet>
<servlet-name>spring</servlet-name>
<servlet-class>
org.springframework.web.servlet.DispatcherServlet
</servlet-class>
<load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
<servlet-name>spring</servlet-name>
<url-pattern>/</url-pattern>
</servlet-mapping>
</web-app>
```

spring-servlet.xml

```
<beansxmlns="http://www.springframework.org/schema/beans"
xmlns:context="http://www.springframework.org/schema/context"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd">
<context:component-scan
base-package="com.springrestexample.controller,com.springrestexample.service"/>
</beans>
```



The RestTemplate class is given by spring framework to access the Restful services in the easy manner.

Test.java

```
package com.springrestclientexample;
import java.util.HashMap;
import java.util.Map;
import org.springframework.web.client.RestTemplate;
publicclass Test{
publicstaticvoid main( String[] args )  {
  String url="http://localhost:8082/SpringRestServerEx/getUserInfo/{userId}";
   RestTemplate restTemplate=new RestTemplate();
   Map<String,Object>map=new HashMap<String, Object>();
map.put("userId",101);
   String jsonUser=restTemplate.getForObject(url,String.class,map);
   System.out.println(jsonUser);
   }
}
```

Response class:

- **Response class is present in javax.ws.rs.core package.**
- Defines the contract between a returned instance and the runtime when an application needs to provide metadata to the runtime.
- An application class can extend this class directly or can use one of the static methods to create an instance of this class .

The following Example is showing how to use Response class as a return type to resource method

```
package com.nareshit.service;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.core.Response;
@Path("hello")
public class HelloWorldService {
        @GET
        @Path("/sayHello/{name}")
        public Response getMsg(@PathParam("name") String name) {
                String output = "Hello  : " + name+"  Welcome ";
                return Response.status(200).entity(output).build();
        }
}
```

The following Example is showing how to use XML format data with Response obj

```
public class StudentService {
        @GET
        @Path("/getStudentDetails/{sid}")
        @Produces(MediaType.APPLICATION_XML)
        public Response getResult(@PathParam("sid) Integer sid){
                Student std=new Student();
                std.setName("sathish");
                std.setSid(sid);
        return Response.status(200).entity(std).build();
        }
}
```

Student.java

```
package com.nareshit.domain;
import javax.xml.bind.annotation.XmlRootElement;
@XmlRootElement
public class Student {
private Integer sid;
private String name;
//setters and getters
}
```

Get HTTP header in JAX-RS

We have two ways to get HTTP request header in JAX-RS:

1. Inject directly with @HeaderParam
2. Programatically via @Context

1. @HeaderParam Example

In this example, it gets the browser "user-agent" from request header.

```
Import javax.ws.rs.GET;
import javax.ws.rs.HeaderParam;
import javax.ws.rs.Path;
import javax.ws.rs.core.Response;
@Path("/users")
public class UserService {
@GET
public Response  getHeadersResult(@HeaderParam("user-agent") String userAgent){
System.out.println("getHeaderResult method is calling and userAgent is :"+userAgent);
return Response.status(200).entity(userAgent).build();
}
```

2) Pragmatically via @Context Annnotation :

For @HeaderParam annotation Alternatively, you can also use @Context to get "javax.ws.rs.core.HttpHeaders" directly, see equivalent example to get browser "user-agent".

```
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.HttpHeaders;
import avax.ws.rs.core.Response;
@Path("/users")
public class UserService {
@GET
@Path("/get")
  @Produces("text/plain")
  public String getHeaders(@Context HttpHeaders headers) {
String userAgent = headers.getRequestHeader("user-agent").get(0);
return  "getHeaders is called, userAgent : " + userAgent;
}
}
```

List all request headers

You can also get list of available HTTP request headers via following code :

```
@Path("/users")
public class UserService {
  @GET
@Path("/getAll")
@Produces("text/plain")
public Response getAllHeaders(@Context HttpHeaders headers){
  MultivaluedMap<String,String> map=headers.getRequestHeaders();
    Set<String> keys=map.keySet();
  for(String key:keys){
          System.out.println("Header Name : "+key);
  List<String> headerValues=headers.getRequestHeader(key);
     for(String headerValue:headerValues){
   System.out.println(headerValue);
     }
  }
          return Response.status(200).build();
}
}
```

Note :

In general, @Context can be used to obtain contextual Java types related to the request or response.

(for example `@Context HttpServletRequest request`).

@CookieParam:

- Cookie is a name-value pair.
- In General Cookie is transferring from Server to client and client to server.
- The Cookie object is creating in Server System but stores in Client System.

We can retrieve the entire cookie by injecting the Cookie with the @CookieParam annotation by providing the cookie name.

```
@GET
@Produces("text/plain")
public String getCookie(@CookieParam("cookie-name")  Cookie cookie){
System.out.println(cookie);
return "OK";
}
```

We can also retrieve just the cookie value by injecting the value with the @CookieParam annotation.

```
@GET
@Produces("text/plain")
public String getCookieValue(@CookieParam("name") String cookie){
System.out.println(cookie);
return "OK";
}
```

Extracting Request Parameters:

Parameters of a resource method may be annotated with parameter-based annotations to get the values from a request obj.

You can extract the following types of parameters for use in your resource class:

- QueryParam
- URI path(PathParam)
- FormParam
- Cookie Param
- Header Param
- MatrixParam

Query parameters are retrieved from the request URI query parameters and are specified by using the javax.ws.rs.QueryParam annotation in the method parameter arguments.

The following example, from the Employee application, demonstrates using of @QueryParam annotation to retrieve query parameters from the requested URL:

```
package com.nareshit.service;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.QueryParam;
import javax.ws.rs.core.Response;
@Path("employee")
public class EmployeeService {
 @GET
@Path("/getEmployeeDetails")
public Response getEmployee(@QueryParam("dept") String department,
```

```
@QueryParam("location") String location){
        String resp = "Query parameters are received. 'dept' value is: "
                    +department+" and location value is: "+location;
          return Response.status(200).entity(resp).build();
    }
}
```

In the above example, if you use "***/employee/getEmployeeDetails?location=hyderabad&dept=finance***" *URI pattern with query* parameters, getEmployee() method will be invoked, and you will get "**Query parameters as 'dept' value is: finance and location value is: hyderabad**" as a response.

To assign default values to method input variables if the query parameters are not available You can use @DefaultValue annotation to specify default value.

```
package com.nareshit.service;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.QueryParam;
import javax.ws.rs.core.Response;
@Path("employee")
public class EmployeeService {
 @GET
@Path("/getEmployeeDetails")
public Response getEmployee (
@DefaultValue("accounts") @QueryParam("dept") String department,
@DefaultValue("bangalore") @QueryParam("location") String location){
        String resp = "Query parameters are received. 'dept' value is: "
                    +department+" and location value is: "+location;
returnResponse.status(200).entity(resp).build();
    }
}
```

In the above example, if you use "**/employee/getEmployeeDetails**" URI pattern, and you will get "**Query parameters are received. 'dept' value is: accounts and location value is: bangalore**" as a response.

Sometimes User-defined Java programming language types may be used as query parameters(OR) primitive Types used as a query parameters (OR) collection types used as a query parameters as per the requirements.

```
package com.nareshit.service;

import javax.ws.rs.GET;

import javax.ws.rs.Path;

import javax.ws.rs.QueryParam;

import javax.ws.rs.core.Response;

@Path("employee")

public class EmployeeService {

 @GET

@Path("/getEmployeeByDeptNo")

@Produces(MediaType.TEXT_PLAIN)

public String getEmployee(@QueryParam("deptNo") intdeptNo ){



return "Employee Name =raja ,Emp No = 101 , salary =2000 ,deptNo="+deptNo;

   }

  @GET

@Path("/getEmployeeByDeptName/{deptName}")

@Produces(MediaType.TEXT_PLAIN)

public String getEmployee(@PathParam("deptName") Employee emp){



return "Employee Name =raja ,Emp No = 101 , salary =2000
,deptName="+emp.getDeptName();

   }

@GET

@Path("/deleteDepartments")
```

```
@Produces(MediaType.TEXT_PLAIN)

public String deleteDepartments(@QueryParam("deptNo")   List<Integer>deptNoList){

System.out.println("List of Deprts :"+deptNoList);

return "Departments deleted ";

   }

}
```

The following code shows how to write Employee class for above example

```
public class Employee{

private String deptName;

public Employee(String deptName) {

this.deptName=deptName;

   }

public void setDeptName(String deptName){
this.deptName=deptName;

}

public String getDeptName(){
return deptName;

}

}
```

The constructor for Employee takes a single String parameter.

Note :Both @QueryParam and @PathParam can be used only on the following Java types:

- All primitive types except char
- All wrapper classes of primitive types except Character
- Any class with a constructor that accepts a single String argument
- Any class with the static method named valueOf(String) that accepts a single String argument

- List<T>, Set<T>, or SortedSet<T>, where *T* matches the already listed criteria. Sometimes, parameters may contain more than one value for the same name. If this is the case, these types may be used to obtain all values

Note :

If @DefaultValue is not used in conjunction with @QueryParam, and the query parameter is not present in the request, the value will be an empty collection for List, Set, or SortedSet; null for other object types; and the default for primitive types.

BeanParameters(By using @BeanParam annotation) (JAX-RS 2.0)):

```
package com.nareshit.bean;
public class EmployeeBean{
@QueryParam("salary")
private double salary;
@QueryParam("deptName")
private String deptName;
@QueryParam("location")
private String location;
//setters and getters
}
```

```
package com.nareshit.service;
@Path("/employee")
public class EmployeeService{
@GET
@Produces(MediaType.TEXT_PLAIN)
@Path("/searchEmployees")
public String searchEmployees(@BeanParam  EmployeeBean   employeeBean){
return "empNo =101 ,empName=rama,salary="+employeeBean.getSalary();
}
}
```

Exception-Handling:

- When we send a request to the resource, the resource sometimes may throws Exceptions.
- If resource throws Exception , JAXRS runtime will convert that exception into an response message with status code as error response code.
- But if we want to return a custom error response rather than a default error response message we need to handle exceptions and should return as a response shown below.

UserNotFoundException.java

```
public class UserNotFoundException extends Exception {
        public UserNotFoundException() {
            super();
          }
         public UserNotFoundException (String msg) {
            super(msg);
          }
public UserNotFoundException (String msg, Exception e) {
super(msg, e);
          }
        }
```

UserService.java

```
@Path("users")
public  class UserService {
 @GET
@Path("/getUser/{userId}")
@Produces(MediaType.APPLICATION_JSON)
public Response getUser(@PathParam("userId")  String userId) throws UserNotFoundException {
if(userId.equals("101")){
User user=new User();
user.setUserId(userId);
user.setName("sathish");
user.setEmail("sathish@gmail.com");
user.setMobile("8888889988");
return Response.status(200).entity(user).type(MediaType.APPLICATION_JSON).build();
  }
 else{
 throw new UserNotFoundException ("User does not exist with id " +     userId);
}
 }
}
```

This above method may throws an appropriate exception when a user cannot be found. However, we still need to create a Handler object to convert this exception into an actual JSON response so we get a nice friendly error message. The class below handles this exception and will convert the error message in the exception into a JSON response. The important annotation you'll see on this class i,e @Provider annotation.

UserNotFoundExceptionHandler :

Caching:

When we opening of a webpage for the first time takes some time, but the second or third time it loads faster. This happens because whenever we visit a webpage for the first time, our browser caches the content and need not have to make a call over the network to render it.

This caching ability of the browser saves a lot of network bandwidth and helps in cutting down the server load.

1.) Browser or local Caches: This is the local in-memory cache of a browser. This is the fastest cache available. Whenever we hit a webpage a local copy is stored in browser and then second time it uses this local copy instead of making a real request over the network.

2.) Proxy Caches: These are pseudo web servers that work as middlemen between browsers and websites.

These servers cache the static content and serve its clients so the client does not have to go to server for these resources.

Content Delivery Networks (CDN) are of similar concept where in they provide proxy caches from their servers , which helps in faster serving the content and sharing the load of servers.

Inconsistency and invalidation: When we are dealing with cache we need to make sure that the stale data is invalidated and it should be consistent. So let's see what all mechanisms are provided by HTTP which can help us make effective use of HTTP caching.

HTTP Headers

Before HTTP 1.1 the only way to control caching behavior was with the help of "expries" header. In this an expiry date can be specified.

But, later more powerful and extensive headers were introduced in HTTP 1.1, this was "cache-control" header. This along with "retag" gave the real power to the applications to control the behavior of caches.

JAX-RS supports these and provides APIs to use them. We will explore how we can leverage caching using HTTP response headers and the support provided by JAX-RS.

Expires Header:

In HTTP 1.0, a simple response header called Expires would tell the browser how long it can cache an object or page. It would be a date in future after which the cache would not be valid.
So, if we made an API call to retrieve data :

```
GET /users/1
The response header would be:
HTTP/1.1 200 OK
Content-Type: application/xml
Expires: Tue, 20 Dec 2016 16:00 GMT
-----
<user id="1">...</users>
```

- This means the XML data is valid until 20th Dec 2016, 16:00 hours GMT.
- JAX-RS supports this header in **javax.ws.rs.core.Response** object.

```
package com.cacheexample.service;

@Path("user")
public class UserService{
@Path("getUser/{uid}")
@GET
@Produces(MediaType.APPLICATION_XML)
public Response getUserXML(@PathParam("uid") Long uid){
User user=null;
if(uid.equals(101)){
user=new User();
user.setUserId(uid);
user.setUserName("Ashok");
user.setEmail("ashok@gmail.com");
}
ResponseBuilder builder = Response.ok(user,MediaType.APPLICATION_XML); //Putting expires
header for HTTP browser caching.
Calendar cal = Calendar.getInstance();
cal.set(2016,12,25,16,0);
builder.expires(cal.getTime());
return builder.build();
}
```

Note : ResponseBuilder object build the Response object

But to support CDNs, proxy caches there was a need for more enhanced headers with richer set of features, having more explicit controls. Hence in HTTP 1.1 few new headers were introduced and Expires was deprecated. Let's explore them.

With Cache-Control :

Cache-Control has a variable set of comma-delimited directives that define who, how and for how long it can be cached. Let's explore few of them:

–**private/public** : these are accessibility directives, private means a browser can cache the object but the proxies or CDNs cannot and public makes it cacheable by all.

-no-cache, no-store,max-age are few others where name tells the story.

JAX-RS provides **javax.ws.rs.core.CacheControl** class to represent this header.

```
package  com.cacheexample.service;
@Path("user")
public class UserService{
@Path("getUser/{uid}")
 @GET
@Produces(MediaType.APPLICATION_XML)
public Response getUserXMLwithCacheControl(@PathParam("uid") Long uid){
User user=null;
if(uid.equals(101)){
user=new User();
user.setUserId(uid);
user.setUserName("Ashok");
user.setEmail("ashok@gmail.com");
}
CacheControl cc = new CacheControl();
cc.setMaxAge(300);//  set  the max-age cache control directive.
cc.setPrivate(true);
ResponseBuilder builder = Response.ok(user,MediaType.APPLICATION_XML);
 builder.cacheControl(cc);
return  builder.build();
}
```

RESTful Web Service Security

Security Areas:

There are two main areas for securities.

Authentication: Process of checking the user, who they claim to be.

Authorization: Process of deciding whether a user is allowed to perform an activity within the application.

Different Authentications :

- *BASIC Authentication*
    - It's simplest of all techniques and probably most used as well. You use login/password forms – it's basic authentication only. You input your username and password and submit the form to server, and application identify you as a user – you are allowed to use the system – else you get error.

- The main problem with this security implementation is that credentials are propagated in a plain way from the client to the server. Credentials are merely encoded with Base64 in transit, but not encrypted or hashed in any way. This way, any sniffer could read the sent packages over the network.

DIGEST Authentication

This authentication method makes use of a hashing algorithms to encrypt the password (called **password hash**) entered by the user before sending it to the server. This, obviously, makes it much safer than the basic authentication method, in which the user's password travels in plain text that can be easily read by whoever intercepts it.

CLIENT CERT Authentication

- This is a mechanism in which a trust agreement is established between the server and the client through certificates. They must be signed by an agency established to ensure that the certificate presented for authentication is legitimate, which is known as CA.
- Using this technique, when the client attempts to access a protected resource, instead of providing a username or password, it presents the certificate to the server. The certificate contains the user information for authentication including security credentials, besides a unique private-public key pair. The server then determines if the user is legitimate through the CA. Additionally, it must verify whether the user has access to the resource. This mechanism must use HTTPS as the communication protocol as we don't have a secure channel to prevent anyone from stealing the client's identity.

Ways to Secure RestFullWeb Services:

You can secure your Restful Web services using one of the following methods to support authentication, authorization, or encryption:

1) Using **web.xml** deployment descriptor to define security configuration.

2) Using the **javax.ws.rs.core.SecurityContext** interface to implement security programmatically.

3) By Applying annotations to your JAX-RS classes.

Securing RESTful Web Services Using web.xml

- You secure RESTful Web services using the **web.xml** deployment descriptor as you would for other Java EE Web applications.

- For example, to secure your RESTful Web service using basic authentication, perform the following steps:

1. Define a **<security-constraint>** for each set of RESTful resources (URIs) that you plan to protect.
2. Use the **<login-config>** element to define the type of authentication you want to use and the security realm to which the security constraints will be applied.
3. Define one or more security roles using the **<security-role>** tag and map them to the security constraints defined in step 1.
4. To enable encryption, add the **<user-data-constraint>** element and set the **<transport-guarantee>** subelement to **CONFIDENTIAL**

Example 1 Securing RESTful Web Services Using Basic Authentication

Step1 : create Root Resource class as for your requirement :

```
package com.nareshit.service;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.QueryParam;
import javax.ws.rs.core.MediaType;
@Path("products")
public class ProductService {
@GET
@Path("/getProductName")
@Produces(MediaType.TEXT_PLAIN)
public String getProductName(
                @QueryParam("pid") Integer pid){
        if(pid!=null && pid.equals(301)){
        return "keyboard";
        }
        return "Product not found";
        }
}
```

Step2:- Configure The Security Configuration in web.xml file as follows

```
<web-app>
<servlet>
<servlet-name>RestServlet</servlet-name>
<servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>
</servlet>
<servlet-mapping>
<servlet-name>RestServlet</servlet-name>
<url-pattern>/*</url-pattern>
</servlet-mapping>
<security-constraint>
<web-resource-collection>
<web-resource-name>Products</web-resource-name>
<url-pattern>/*</url-pattern>
<http-method>GET</http-method>
<http-method>POST</http-method>
</web-resource-collection>
<auth-constraint>
<role-name>ADMIN</role-name>
```

Step3:- Set username and password and Role's in Tomcat Server

Changes in Tomcat Server (tomcat-users.xml)

As stated above we are using Tomcat 7. Now for the user to be authenticated we will specify the role 'ADMIN' (*which is the role chosen in our* web.xml *above*) and some username and password in our container. This username and password will have to be supplied to access the restricted resource.

tomcat-users.xml

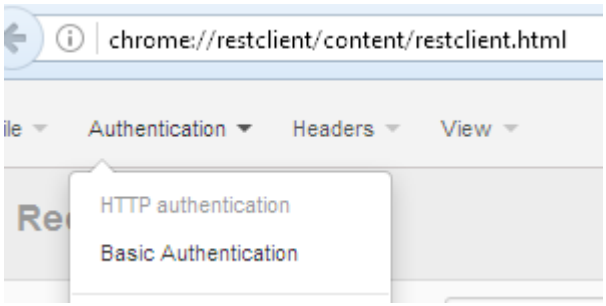
```
<tomcat-users>
<role rolename="manager-gui"/>
<role rolename="admin-gui"/>
<role rolename="admin"/>
<role rolename="customer"/>
<user username="customer" password="customer" roles="customer"/>
<user username="admin" password="admin" roles="admin"/>
<user username="admin" password="admin" roles="admin-gui,manager-gui"/>
</tomcat-users>
```

Step4:- Deploy the Project into the server

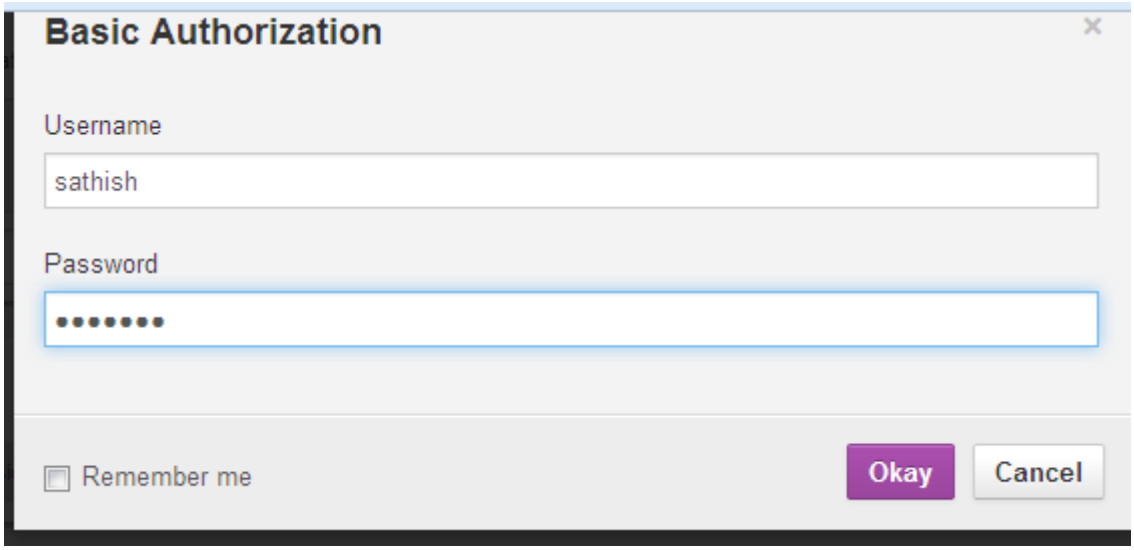
Note : Test the above Service By using Any RestClient tool.

While sending the request set username and password as follows.

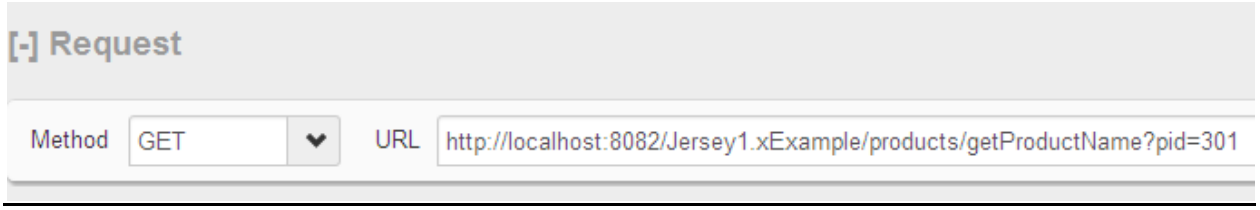
First Click on Authentication in the client tool→Basic Authentication



Enter UserName and password



Enter the URL and send



Output :

[-] Response

Response Headers | Response Body (Raw) | Response Body (Highlight) | Response Body (Preview)

```
1.  Status Code        : 200 OK
2.  Cache-Control      : private
3.  Content-Type       : text/plain
4.  Date               : Fri, 16 Dec 2016 13:48:50 GMT
5.  Expires            : Thu, 01 Jan 1970 05:30:00 IST
6.  Server             : Apache-Coyote/1.1
7.  Transfer-Encoding  : chunked
```

2) Securing RESTful Web Services Using SecurityContext

The **javax.ws.rs.core.SecurityContext** interface provides access to security-related information for a request. The **SecurityContext** provides functionality similar to **javax.servlet.http.HttpServletRequest**, enabling you to access the following security-related information:

- **java.security.Principal** object containing the name of the user making the request.
- Authentication type used to secure the resource, such as BASIC_AUTH, FORM_AUTH, and CLIENT_CERT_AUTH.
- Whether the authenticated user is included in a particular role.
- Whether the request was made using a secure channel, such as HTTPS.

You access the **SecurityContext** by injecting an instance into a class field, setter method, or method parameter using the **javax.ws.rs.core.Context** annotation.

The following shows how to inject an instance of **SecurityContext** into the **sc** method parameter using the **@Context** annotation, and check whether the authorized user is included in the **admin** role before returning the response.

Example 2 Securing RESTful Web Service Using SecurityContext

```
package com.nareshit.service;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.QueryParam;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.MediaType;
```

```
import javax.ws.rs.core.SecurityContext;
@Path("products")
public class ProductService {
@GET
@Path("/getProductName")
@Produces(MediaType.TEXT_PLAIN)
public String getProductName(
                    @QueryParam("pid") Integer pid,
                    @Context SecurityContext sc) {
          if (sc.isUserInRole("admin")){
          if(pid!=null && pid.equals(301)){
          return "keyboard";
          }
          else{
          return "Product Not Found";
          }
 }
}
}
```

3) Securing RESTful Web Services Using Annotations

The **javax.annotation.security** package provides annotations, defined in Table 1, that you can use to secure your RESTful Web services.

Table 1 Annotations for Securing RESTful Web Services

Annotation	Description
DeclareRoles	Declares roles.
DenyAll	Specifies that no security roles are allowed to invoke the specified methods.
PermitAll	Specifies that all security roles are allowed to invoke the specified methods.
RolesAllowed	Specifies the list of security roles that are allowed to invoke the methods in the application.
RunAs	Defines the identity of the application during execution in a J2EE container.

In The following Example getProductName method is annotated with the **@RolesAllowed** annotation to override the default and only allow users that belong to the **ADMIN** security role.

Example 3 Securing RESTful Web Service Using Annotations(javax.annotation.security)

```
package com.nareshit.service;
import javax.annotation.security.RolesAllowed;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.QueryParam;
import javax.ws.rs.core.MediaType;
@Path("products")
@RolesAllowed({"admin","customer"})
public class ProductService {
@GET
@Path("/getProductName")
@Produces(MediaType.TEXT_PLAIN)
@RolesAllowed("admin")
public String getProductName(
        @QueryParam("pid") Integer pid) {
        if(pid!=null && pid.equals(301)){
        return "keyboard";
        }
        else{
        return "Product Not Found";
        }
}
}
```

Realm :

- A realm is a repository of user information; it is an abstraction of the data store – text file, JDBC database or a JNDI resource. This has the following information: username, password and the roles which are assigned to the users.
- Both of the authentication and authorization make up the security policy of a server. Tomcat uses realms to implement container-managed security and enforce specific security policies.

Conculsion :

RESTful Services:

REST stands for Representational State Transfer. REST was a term coined by Roy Fielding in his doctoral dissertation. It is an architecture style for for creating network based applications. Key properties of REST are client-server communication, stateless protocol, cacheable, layered implementation and uniform interface

As RESTful web services work with HTTP URLs Paths so it is very important to safeguard a RESTful web service in the same manner as a website is be secured. Following are the best practices to be followed while designing a RESTful web service.

- **Validation** - Validate all inputs on the server. Protect your server against SQL or NoSQL injection attacks.
- **Session based authentication** - Use session based authentication to authenticate a user whenever a request is made to a Web Service method.
- **No sensitive data in URL** - Never use username, password or session token in URL , these values should be passed to Web Service via POST method.
- **Restriction on Method execution** - Allow restricted use of methods like GET, POST, DELETE. GET method should not be able to delete data.
- **Validate Malformed XML/JSON** - Check for well formed input passed to a web service method.
- **Throw generic Error Messages** - A web service method should use HTTP error messages like 403 to show access forbidden etc.

S.N.	HTTP Code & Description
1	**200** **OK**, shows success.
2	**201** **CREATED**, when a resource is successful created using POST or PUT request. Return link to newly created resource using location header.
3	**204** **NO CONTENT**, when response body is empty for example, a DELETE request.
4	**304** **NOT MODIFIED**, used to reduce network bandwidth usage in case of conditional GET requests. Response body should be empty. Headers should have date, location etc.
5	**400** **BAD REQUEST**, states that invalid input is provided e.g. validation error, missing data.
6	**401** **UNAUTHORIZED**, states that user is using invalid or wrong authentication token.
7	**403** **FORBIDDEN**, states that user is not having access to method being used for example, delete access without admin rights.
8	**404** **NOT FOUND**, states that method is not available.
9	**409** **CONFLICT**, states conflict situation while executing the method for example, adding duplicate entry.
10	**500** **INTERNAL SERVER ERROR**, states that server has thrown some exception while executing the method.

JAX-RS Annotations:

Following are the commonly used annotations to map a resource as a web service resource.

S.N.	Annotation & Description
1	**@Path** Relative path of the resource class/method.
2	**@GET** HTTP Get request, used to fetch resource.
3	**@PUT** HTTP PUT request, used to create resource.
4	**@POST** HTTP POST request, used to create/update resource.
5	**@DELETE** HTTP DELETE request, used to delete resource.
6	**@HEAD** HTTP HEAD request, used to get status of method availability.
7	**@Produces** States the HTTP Response generated by web service, for example APPLICATION/XML, TEXT/HTML, APPLICATION/JSON etc.
8	**@Consumes** States the HTTP Request type, for example application/x-www-form-urlencoded to accept form data in HTTP body during POST request.
9	**@PathParam** Binds the parameter passed to method to a value in path.
10	**@QueryParam** Binds the parameter passed to method to a query parameter in path.
11	**@MatrixParam** Binds the parameter passed to method to a HTTP matrix parameter in path.
12	**@HeaderParam** Binds the parameter passed to method to a HTTP header.
13	**@CookieParam** Binds the parameter passed to method to a Cookie.
14	**@FormParam** Binds the parameter passed to method to a form value.
15	**@DefaultValue** Assigns a default value to a parameter passed to method.

POSTMAN

Introduction

Postman is a Chrome add-on and Mac application which is used to fire requests to an API. It is very lightweight and fast. Requests can be organized in groups, also tests can be created with verifications for certain conditions on the response. With its features, it is very good and convenient API tool. It is possible to make different kinds of HTTP requests – GET, POST, PUT, PATCH and DELETE. It is possible to add headers to the requests.

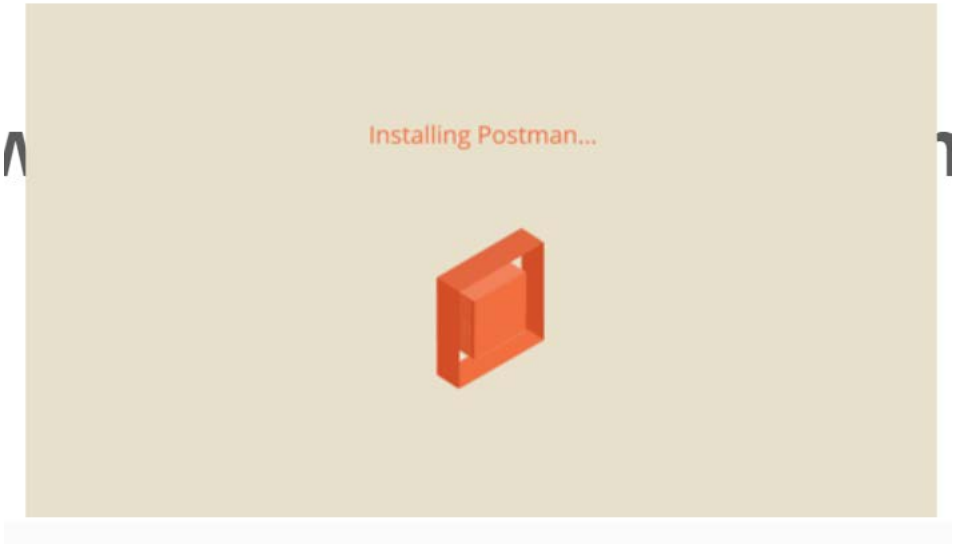
Installation

Step-1 → Open Google Chrome and type Postman → Click on below highlighted link

Step-2 → After clicking on "https://www.getpostman.com/apps" this URL below screen will be displayed

Step-3 → Based on your operating system click on "Download" button (below Screen will be displayed)

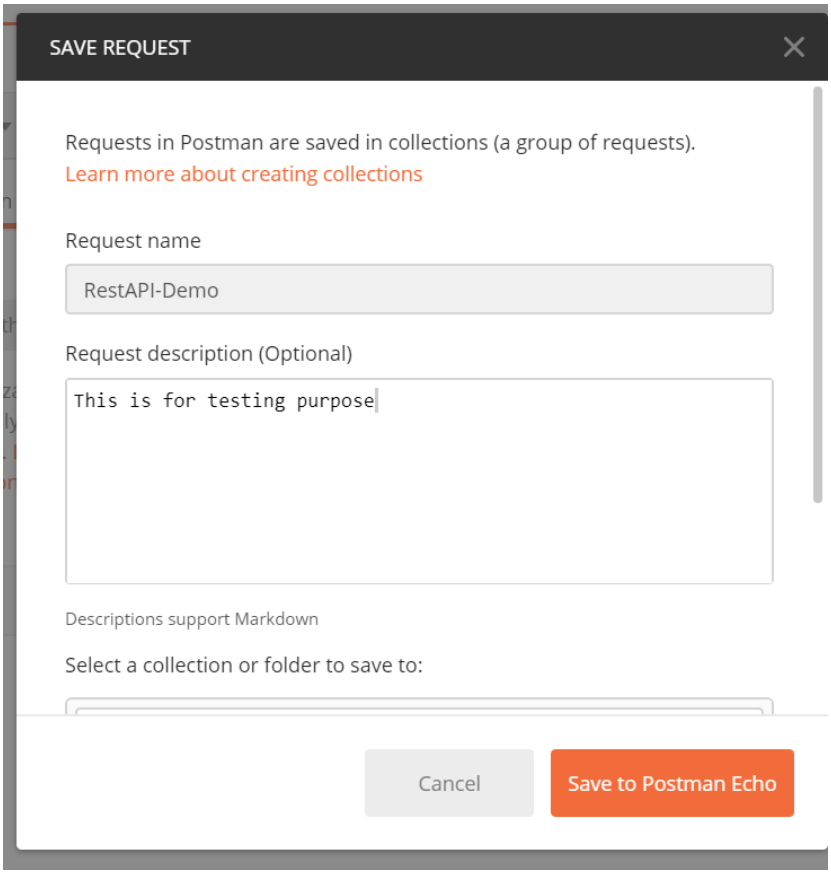
Stpe-4 → Install downloaded exe file (Below screen will be displayed)



Step-5 → It displays below screen (Create a free account)

Step-6 → After Account creation Login into Postman using given credentials (below screen will be displayed)

Step-7 -> From the above screen click on Request and enter details (Selcet PostMan Echo) and click on "Save to Postman Echo" button



Step-8 → We can send http request to Rest API using request URL from below screen

In this screen we can use request type like GET, POST, PUT and Delete

POSTMAN Screen shot for POST request

Spring Integration with REST

Introduction

RESTful web services are built to work best on the Web. Representational State Transfer (REST) is an architectural style that specifies constraints, such as the uniform interface, that if applied to a web service induce desirable properties, such as performance, scalability, and modifiability that enable services to work best on the Web. In the REST architectural style, data and functionality are considered resources and are accessed using Uniform Resource Identifiers (URIs), typically links on the Web. The resources are acted upon by using a set of simple, well-defined operations. The REST architectural style constrains an architecture to a client/server architecture and is designed to use a stateless communication protocol, typically HTTP. In the REST architecture style, clients and servers exchange representations of resources by using a standardized interface and protocol.

The following principles encourage RESTful applications to be simple, lightweight, and fast:

Resource identification through URI: A RESTful web service exposes a set of resources that identify the targets of the interaction with its clients. Resources are identified by URIs, which provide a global addressing space for resource and service discovery. See the @Path Annotation and URI Path Templates for more information.

Uniform interface: Resources are manipulated using a fixed set of four create, read, update, delete operations: PUT, GET, POST, and DELETE. PUT creates a new resource, which can be then deleted by using DELETE. GET retrieves the current state of a resource in some representation. POST transfers a new state onto a resource. See Responding to HTTP Methods and Requests for more information.

Self-descriptive messages: Resources are decoupled from their representation so that their content can be accessed in a variety of formats, such as HTML, XML, plain text, PDF, JPEG, JSON, and others. Metadata about the resource is available and used, for example, to control caching, detect transmission errors, negotiate the appropriate representation format, and perform authentication or access control. See Responding to HTTP Methods and Requests and Using Entity Providers to Map HTTP Response and Request Entity Bodies for more information.

Stateful interactions through hyperlinks: Every interaction with a resource is stateless; that is, request messages are self-contained. Stateful interactions are based on the concept of explicit state transfer. Several techniques exist to exchange state, such as URI rewriting, cookies, and hidden form fields. State can be embedded in response messages to point to valid future states of the interaction. See Using Entity Providers to Map HTTP Response and Request Entity Bodies and “Building URIs” in the JAX-RS Overview document for more information.

RESTful Web Services utilize the features of the HTTP Protocol to provide the API of the Web Service. It uses the HTTP Request Types to indicate the type of operation:

GET: Retrieve / Query of existing records.

POST: Creating new records.

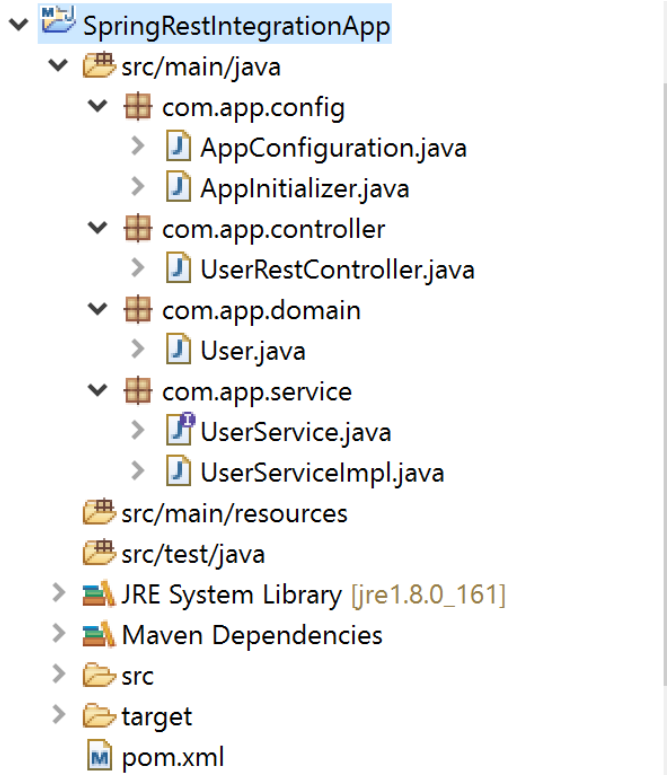
DELETE: Removing records.

PUT: Updating existing records.

Using these 4 HTTP Request Types a RESTful API mimics the CRUD operations (Create, Read, Update & Delete). REST is stateless, each call the to a RESTful Web Service is completely stand-alone, it has no knowledge of previous requests.

Below REST application performs CURD operations using Spring Service class

Step 1: Create Maven Web Project



Step 2: Configure maven dependencies in project pom.xml file

```
<properties>
        <springframework.version>4.3.0.RELEASE</springframework.version>
        <jackson.library>2.7.5</jackson.library>
</properties>
<dependencies>
        <dependency>
                <groupId>org.springframework</groupId>
                <artifactId>spring-core</artifactId>
                <version>${springframework.version}</version>
        </dependency>
        <dependency>
                <groupId>org.springframework</groupId>
                <artifactId>spring-web</artifactId>
                <version>${springframework.version}</version>
        </dependency>
        <dependency>
                <groupId>org.springframework</groupId>
                <artifactId>spring-webmvc</artifactId>
                <version>${springframework.version}</version>
        </dependency>
        <dependency>
                <groupId>javax.servlet</groupId>
                <artifactId>javax.servlet-api</artifactId>
                <version>3.1.0</version>
        </dependency>
        <dependency>
                <groupId>com.fasterxml.jackson.core</groupId>
                <artifactId>jackson-databind</artifactId>
                <version>${jackson.library}</version>
        </dependency>
</dependencies>
```

Step 3: Create Domain class for Storing and Retrieving the data (User.java)

User.java

```
package com.app.domain;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement(name = "user")
@XmlAccessorType(XmlAccessType.FIELD)
public class User {

    private Integer userid;
    private String username;
    private String email;
    private String gender;

    //setters & getters

    //toString( )

}
```

Step 4: Create UserService.java class to perform Business operations

```
package com.app.service;

@Service(value = "service")
public class UserServiceImpl implements UserService {

	private static Map<Integer, User> usersData = new HashMap<Integer, User>();

	public boolean add(User user) {
		if (usersData.containsKey(user.getUserid())) {
			return false;
		} else {
			usersData.put(user.getUserid(), user);
			return true;
		}
	}
	public User get(String uid) {
		System.out.println(usersData);
		if (usersData.containsKey(Integer.parseInt(uid))) {
			return usersData.get(Integer.parseInt(uid));
		}
		return null;
	}
	public boolean update(String uid, User user) {
		if (usersData.containsKey(Integer.parseInt(uid))) {
			usersData.put(Integer.parseInt(uid), user);
			return true;
		}
		return false;
	}
	public boolean delete(String uid) {
		if (usersData.containsKey(Integer.parseInt(uid))) {
			usersData.remove(usersData.get(Integer.parseInt(uid)));
			return true;
		}
		return false;
	}
}
```

UserServiceImpl.java

Step 5: Create RestController (UserRestController.java)

```
package com.app.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RestController;

import com.app.domain.User;
import com.app.service.UserService;

@RestController
public class UserRestController {

	@Autowired(required = true)
	private UserService service;

	@RequestMapping(value = "/add", method = RequestMethod.POST, consumes = { "application/xml",
"application/json" })
	public @ResponseBody String addUser(@RequestBody User user) {
		boolean isAdded = service.add(user);
		if (isAdded) {
			return "User Added successfully";
		} else {
			return "Failed to Add the User..!";
		}
	}
```

```
	@RequestMapping(value = "/get", produces = { "application/xml", "application/json" }, method =
RequestMethod.GET)
	@ResponseBody
	public User getUserById(@RequestParam(name = "uid") String uid) {
		System.out.println("Getting User with User Id : " + uid);
		User user = service.get(uid);
		return user;
	}

	@RequestMapping(value = "/update", method = RequestMethod.PUT,
			consumes = { "application/xml", "application/json" }
	)
	public @ResponseBody String update(@RequestParam("uid") String uid, @RequestBody User user) {
		boolean isAdded = service.update(uid, user);
		if (isAdded) {
			return "User updated successfully";
		} else {
			return "Failed to update the User..!";
		}
	}

	@RequestMapping(value = "/delete", method = RequestMethod.DELETE)
	public @ResponseBody String delete(@RequestParam("uid") String uid) {
		boolean isAdded = service.delete(uid);
		if (isAdded) {
			return "User Deleted successfully";
		} else {
			return "Failed to Delete the User..!";
		}
	}

	public void setService(UserService service) {
		this.service = service;
	}
}
```

Step 6: Create AppConfig and AppInitiazer classes

AppConfiguration.java

AppInitializer.java

```
AppInitializer.java
package com.app.config;

import org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;

public class AppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Class<?>[] getRootConfigClasses() {
        return new Class[] { AppConfiguration.class };
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        return null;
    }

    @Override
    protected String[] getServletMappings() {
        return new String[] { "/rest/*" };
    }

}
```

Step 7: Deploy the Application into server (I have used Apache Tomcat 8.0)

Step 8: Test the Application using POSTMAN plugin in Google Chrome

Testing Add User

URI: http://localhost:6060/SpringRestIntegrationApp/rest/add

Method Type: POST

Consumes: {application/xml, application/json}

Produces: text/plain

Request Body Data: In XML Format

```
<? xml version="1.0" encoding="UTF-8"?>
<user>
  <userid>101</userid>
  <username>Raju</username>
  <gender>Male</gender>
  <email>Raj@gmail.com</email>
</user>
```

POSTMAN Screenshot

Testing Fetch User

URI: http://localhost:6060/SpringRestIntegrationApp/rest/get?uid=101

Method Type: GET

Input: Request Parameter (? uid=101)

Produces: {application/xml, application/json}

POSTMAN Screenshot

Testing Update User

URL: http://localhost:6060/SpringRestIntegrationApp/rest/update?uid=101

Method Type: PUT

Input in URL: User Id (Request Parameter)? uid=101

Consumes: {application/xml, application/json}

Produces: text/plain

Request Body Data: in XML format

```
<? xml version="1.0" encoding="UTF-8"?>
<user>
  <userid>101</userid>
  <username>Ashok</username>
  <gender>Male</gender>
  <email>ashok.b@gmail.com</email>
</user>
```

POSTMAN Screenshot

Testing Delete User

URL: http://localhost:6060/SpringRestIntegrationApp/rest/delete?uid=101

Method Type: DELETE

Input: Request Parameter (? uid=101)

Produces: text/plain

POSTMAN Screenshot

=== Happing Learning ====